Hannes Bochmann

Analyse des Standes der Technik im Bereich Testen von Software und Etablierung von ersten qualitätssichernden Maßnahmen am Beispiel eines komplexen Webportals auf Basis des WCMS TYPO3

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Elektro- und Informationstechnik

Mittweida, 2010

Hannes Bochmann

Analyse des Standes der Technik im Bereich Testen von Software und Etablierung von ersten qualitätssichernden Maßnahmen am Beispiel eines komplexen Webportals auf Basis des WCMS TYPO3

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Elektro- und Informationstechnik

Mittweida, 2010

Erstprüfer: Prof. Dr. Mario Geißler
Zweitprüfer: Dipl.-Inf. (FH) René Nitzsche

Vorgelegte Arbeit wurde verteidigt am: 21.02.2011

Bibliographische Beschreibung

Bochmann, Hannes:

Analyse des Standes der Technik im Bereich Testen von Software und Etablierung von ersten qualitätssichernden Maßnahmen am Beispiel eines komplexen Webportals auf Basis des WCMS TYPO3

69 Seiten, 19 Abbildungen, 2 Tabellen, 1 Anlage

Mittweida, Hochschule Mittweida (FH), Fakultät Elektro- und Informationstechnik

Diplomarbeit, 2010

Referat

Das MedienKombinat GmbH verwendet derzeit kein werkzeuggestütztes Modell zur Qualitätssicherung. Maßnahmen, wie beispielsweise automatisiertes Testen, sind bisher nicht in den Softwareentwicklungsprozess integriert. Vor allem gegen Ende eines Projektes wenden Entwickler daher zunehmend Zeit für Fehlerkorrekturen auf, was wiederum zu einer geringeren Produktivität führt. Zudem können unentdeckte Fehler fatale Auswirkungen zur Folge haben, wenn sie auf Grund von unzureichenden manuellen Tests in der Software verbleiben. Im Rahmen dieser Diplomarbeit werden geeignete qualitätssichernde Maßnahmen anhand eines komplexen Webportals evaluiert, erprobt und eine weitergehende Konzeptionierung für zukünftige Projekte erarbeitet. Dadurch sollen Fehler frühzeitig identifiziert werden, um die Entwicklungsgeschwindigkeit zu erhöhen und Kosten zu reduzieren.

Ziel dieser Diplomarbeit ist es, einen Testprozess in den Entwicklungsprozess zu integrieren, um die Qualität der produzierten Software langfristig zu steigern.

Danksagung

Vertrauen ist die Basis jeder erfolgreichen Zusammenarbeit. Es ist mir deshalb ein besonderes Vergnügen, mich an dieser Stelle gebührend bei all jenen zu bedanken, die mich auf meinem Weg zum erfolgreichen Diplom begleitet haben.

An erster Stelle möchte ich mich bei meinem Referenten Prof. Dr. Mario Geißler bedanken, der stets für mich ansprechbar war und mir die Freiheit gelassen hat, die Arbeit nach eigenen Vorstellungen zu entwickeln.

Bedanken möchte ich mich auch bei allen Mitarbeitern der MedienKombinat GmbH, besonders bei meinem unternehmerischen Betreuer Herrn Dipl. Inf. (FH) René Nitzsche, der mir dieses Thema anvertraute und somit die Möglichkeit bot, diese Arbeit anzufertigen.

Darüber hinaus möchte ich meine Mutter und meine Freundin Katharina dankend erwähnen, die mich stets unterstützten.

Inhaltsverzeichnis

Sprachliche Gleichbehandlung der Geschlechter

Um die Arbeit leserfreundlich zu gestalten, wurde auf eine durchgehende Nennung beider Geschlechter verzichtet. Wo nur die männliche oder weibliche Form verwendet wird, kann davon ausgegangen werden, dass immer auch das andere Geschlecht gemeint ist.

Glossar

Betriebsmittel umfassen Hardware Konfigurationen wie z.B. Festplatten, Arbeitsspeicher oder Drucker aber auch darüber hinaus benötigte Software.

„Beim **Validieren** bewertet der Tester, ob ein (Teil-)Produkt eine festgelegte (spezifizierte) Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich bzw. nützlich ist.“ (vgl. [Spi05], S. 41)

„Verifikation ist […] auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu einer direkten Spezifikation (Phaseneingangsdokument) nachweisen.“ (vgl. [Spi05], S. 41)

„Im **Unit-Test** werden kleinere Programmteile in Isolation von anderen Programmteilen getestet. Die Granularität der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen.“ (vgl. [Wes06], S. 21)

Ein **Build-Prozess** (von englisch *to build* „bauen“) bezeichnet in der Programmierung einen Vorgang, durch den ein fertiges Anwendungsprogramm automatisch erzeugt wird.

Deployment sind Prozesse zur Installation von Software auf Anwender-PCs oder Servern in Betrieben.

PHPMD ist ein Werkzeug, welches PHP Quellcode auf Probleme prüft, wie zu komplizierten Code, nicht verwendete Parameter und ähnliches.

PHPDoc ist ein Werkzeug zur automatisierten Erstellung einer Dokumentation und facto Standard im PHP Umfeld.

SVN ist eine freierhältliche Software zur Versionsverwaltung von Dateien.

Das **Wasserfallmodell** ist ein lineares Vorgehensmodell in der Softwareentwicklung bestehend aus den Phasen Initialisierung, Analyse, Entwurf, Realisierung, Einführung und Nutzung. Dabei werden die Phasen von den Ergebnissen vorhergehender Phasen beeinflusst.

MVC oder auch Model View Controller ist ein Softwareentwicklungsmuster bestehend aus dem Datenmodell, der Präsentation der Daten und der Programmsteuerung.

Abbildungsverzeichnis

Tabellenverzeichnis

Einleitung

„Fehler sind nützlich, aber nur, wenn man sie schnell findet."

John Maynard Keynes, Mathematiker (1883 – 1946)

Qualitätssicherung in Softwareprojekten ist ein noch nicht allzu weit verbreitetes Thema und wird gerne von vielen Entwicklerteams übergangen. Dieser Umstand ist verwunderlich da Beispiele wie die Bluescreens in Windows Betriebssystemen zeigen, dass minderwertige Qualität in Software schwerwiegende Folgen haben kann. In der Automobilbranche ist es beispielsweise undenkbar für Ingenieure, Projekte ohne Qualitätssicherung durchzuführen. Da u.U. Menschenleben gefährdet werden können, wird nichts dem Zufall überlassen. Klaus Zumwinkel äußerte einmal die passenden Worte *„Qualität ist das Gegenteil des Zufalls."* Auch wenn Softwareprojekte bzw. darin unerwartet auftretende Fehler nicht unbedingt Menschenleben gefährden, handelt es sich doch meist um Transaktionen von sensiblen Geschäftsdaten. Fehler in diesem Bereich können einen immensen finanziellen Schaden nach sich ziehen und zudem zu einem erheblichen Imageverlust führen. Die folgenden Beispiele illustrieren diesen Sachverhalt: (vgl. [URL:Glaser])

1962 wurde ein auf Papier geschriebenes mathematisches Symbol fehlerhaft in den Steuercode einer Rakete integriert. Dies führte zum Verlust der Raumsonde Mariner I, die sich kurz nach dem Start selbst zerstörte. Durch Exekution von Tests nach der Integration der Formel, wäre der Verlust sicherlich zu verhindern gewesen.

1987 löste ein Softwarefehler den Crash der New Yorker Börse aus, wodurch ein Kursverlust des Dow-Jones-Indexes von 20 Prozent ausgelöst wurde. Zu dieser Zeit wurde schon jeder fünfte Aktienhandel durch Computerprogramme abgewickelt. Nutzten allerdings zu viele die gleiche Software mit ähnlichen Strategien, konnte in bestimmten Fällen das Angebot oder die Nachfrage fehlen, was zu extremen und künstlichen Kursschwankungen führte.

Neben diesen Aspekten ist es weiterhin nicht zu unterschätzen, welch zusätzlichen Zeitaufwand es bedeutet, wenn durch stundenlanges Debuggen nach Fehlern im Quellcode gesucht wird, der schon länger nicht mehr gesehen wurde und der zudem auch noch schlecht dokumentiert ist. Aber nicht nur das Beheben von Fehlern, sondern auch das Weiterentwickeln, und Software wird immer weiter entwickelt, sind ohne Qualitätssicherung erheblich erschwert. Peter Glaser (vgl. [URL:Glaser]) beschreibt Softwarefehler als

„Institutionalisierte menschliche Schwächen“. Genau an diesen Schwächen setzt Qualitätssicherung an, um günstige Voraussetzungen für den Erfolg eines Softwareprojektes zu schaffen und Zufälle weitestgehend zu verhindern. Es gibt zwar keine mathematische Formel um zu beweisen, dass Software fehlerfrei ist, aber Qualitätssicherung oder genauer gesagt das Testen gibt klare Indizien.

Qualitätssicherung bedeutet zwar, besonders in der Einführungsphase, einen erhöhten Arbeits- und evtl. auch Kostenaufwand, der sich aber speziell bei größeren Projekten rentieren wird, da alle Anforderungen erfüllt und Fehler unmittelbar behoben werden, womit sich Nachkorrekturen reduzieren. Durch verschiedene Feedback-Mechanismen erhalten alle Projektbeteiligten zu jedem Zeitpunkt einen exakten Überblick in welchem Zustand sich das Projekt befindet und es kann früh genug korrigierend eingegriffen werden. Nicht zu vergessen ist das gestärkte Vertrauen in die eigene Arbeit und das der Auftraggeber.

Die größte Bedeutung fällt hierbei dem Testen zu. Dieser Aspekt der Qualitätssicherung bildet das Hauptaugenmerk dieser Arbeit, wobei weitere Aspekte wie Software-Reviews oder die Dokumentation nur kurz genannt oder angerissen werden.

1 Ausgangssituation

Das MedienKombinat GmbH ist eine Online-Agentur und ein Softwarehaus mit Standorten im sächsischen Chemnitz und der Hauptstadtregion Berlin/Potsdam. Das Mitarbeiter-Team ist interdisziplinär aufgestellt, um leistungsfähige digitale Kommunikationslösungen für international tätige Konzerne und mittelständische Unternehmen, sowie für die öffentliche Verwaltung zu realisieren.

Das Vorgehen an der Schnittstelle zwischen Marketing und Informationstechnologie ermöglichst es, Leistungen aus den Geschäftsbereichen Consulting, Kreation, Web Development und Online Marketing medienübergreifend und effektiv zur Realsierung von Kundenprojekten zu kombinieren.

Darüber hinaus ist das MedienKombinat Mitglied im Verband Digitale Wirtschaft und Förderer von Open Source Software-Lösungen, was im weiteren Verlauf eine wichtige Rolle spielt.

Die Bevorzugung von Open Source Software-Lösungen macht sich im Besonderen durch die Wahl der eingesetzten Technologien bemerkbar. Für den eCommerce Bereich wird durchgehend Magento, für kleinere Projekte das CMS Joomla und für größere Projekte das CMS TYPO3 verwendet.

Für diese Arbeit hat TYPO3 besondere Bedeutung, da das Projekt, für welches erste qualitätssichernde Maßnahmen erprobt und eingeführt werden, auf diesem aufsetzt.

TYPO3 ist ein lizenzkostenfrei erhältliches, quelloffenes Enterprise Web Content Management System, zugeschnitten auf die Bedürfnisse von mittelständischen Unternehmen, Konzernen und die öffentliche Verwaltung für die Verwendung als Internet-, Intranet- oder Extranet-Anwendung. Die Basis bildet PHP im Verbund mit einer beliebigen Datenbank, vorzugsweise MySQL. Das System wurde ursprünglich 1997 vom Dänen Kasper Skårhøj entwickelt. TYPO3 wird heute jedoch von einer professionellen Entwicklergemeinde, mehrheitlich bestehend aus professionell arbeitenden Agenturen und Dienstleistern, und der TYPO3-Association ständig weiterentwickelt und optimiert. Damit ist die Fortentwicklung des Systems in einer Geschwindigkeit möglich, die von einzelnen System- oder Softwareanbietern nicht realisiert werden kann.

Im Gegensatz zu kommerziellen Content Management Systemen fallen bei TYPO3 keine Lizenzkosten an. Der Nutzer kann somit Kosten sparen und gleichzeitig die Qualität eines professionellen Content Management Systems nutzen.

TYPO3 kann mittels einer großen Auswahl an quelloffen verfügbaren Extensions um zusätzliche Geschäftslogik erweitert werden. Das System ermöglicht es, durch diese modulare Architektur passgenaue Individualentwicklungen zu integrieren. Die Entwicklung dieser Erweiterungsmodule ist eines der Geschäftsbereiche von der MedienKombinat GmbH.

Der Name des Projekts, für welches der Stand der Technik im Bereich des Testens analysiert wird und die damit verbundene Etablierung von ersten qualitätssichernden Maßnahmen, lautet Hoga Jobportal. Hoga steht dabei für Hotel und Gastronomie. Dies ist ein Webportal dessen Entwicklung zu Beginn dieser Diplomarbeit bereits begonnen hatte. Eine genauere Beschreibung zu dem Projekt ist in Kapitel 3 zu finden.

Bisher gab es weder für die entwickelten TYPO3-Extensions, noch für die später sichtbare Oberfläche von Projekten innerhalb des MedienKombinats werkzeuggestützte qualitätssichernde Maßnahmen. Dementsprechend sollen beispielsweise die Möglichkeiten, einfach und schnell automatisiert zu testen, evaluiert und erprobt werden, was eine geringere Fehlerrate und niedrigere Kosten zur Folge hat.

2 Grundlagen des Testens

Soll Qualitätssicherung bzw. das Testen erfolgreich sein, müssen gewisse Regeln befolgt, von allen Projektbeteiligten angenommen, genau geplant und strukturiert um gesetzt werden. Joseph Weizenbaum schrieb schon 1976 (vgl. [Wei78])

„Die meisten gegenwärtig verfügbaren Computerprogramme, vor allem die umfangreichsten und wichtigsten unter ihnen, sind nicht ausreichend theoretisch fundiert."

Und bis heute hat sich nicht viel daran geändert. Die für das Testen notwendigen Grundlagen werden in diesem Kapitel erläutert.

2.1 Normen und Qualitätsmerkmale

Nach (vgl. [Wes06], S. 4) definiert sich der Wert von Software durch *„funktionale Qualität im Hinblick auf Funktionalität und Fehlerfreiheit für die einwandfreie Benutzbarkeit einer Software"*, welche durch die funktionalen Anforderungen bestimmt wird und *„strukturelle Qualität mit Hinblick auf Design und Codestruktur für die nahtlose Weiterentwicklung einer Software"*, welche zu einem Großteil durch die nicht-funktionalen Anforderungen bestimmt wird. Diese Anforderungen müssen verifiziert und validiert werden und in Kapitel 2.2.3 näher betrachtet.

2.1.1 Software-Qualitätsbegriff nach ISO/IEC 9126

Da der Begriff der Qualität oftmals für eine unpräzise und meist subjektive Beurteilung eines Gegenstandes verwendet wird (vgl. [Tra96], S.25), reicht die Definition von funktionaler und struktureller Qualität nicht aus.

Die ISO/IEC 9126 ist ein Modell um Software-Qualität sicherzustellen und definiert diese genau wie die DIN 55350-11 als Gesamtheit von Eigenschaften und Merkmalen eines Software-Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.

Im Modell der ISO/IEC 9126 bilden 6 Grundmerkmale die Basis um Software hinsichtlich ihrer Qualität bewerten zu können.

Das Merkmal der Funktionalität umfasst alle Charakteristika, welche die Erfüllung der geforderten Anforderungen an die Software, zum Beispiel durch die Antwort auf Eingaben

durch den Benutzer, beschreiben. Teilmerkmale hiervon sind Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit und Ordnungsmäßigkeit.

Die Zuverlässigkeit beschreibt die Fähigkeit von Software ihr Leistungsniveau über einen klar definierten Zeitraum und unter festgelegten Bedingungen aufrecht zu erhalten und auch über diese Grenzen hinaus ein akzeptables Verhalten aufzuweisen. Diese definiert sich über Reife, Fehlertoleranz, Robustheit und Wiederherstellbarkeit.

Besonders bei Software mit Benutzer-Interaktion spielt es eine entscheidende Rolle für die Akzeptanz der Software, inwiefern der Benutzer diese verstehen, erlernen und anwenden kann. Die Bewertung kann für verschiedene Benutzergruppen unterschiedlich ausfallen. Die Teilmerkmale der Benutzbarkeit sind dementsprechend Verständlichkeit, Erlernbarkeit, Bedienbarkeit aber auch Attraktivität.

Es ist nicht zu unterschätzen, welch immense Betriebsmittel Software benötigen kann. Aus diesem Grund zeichnet sich Software-Qualität ebenfalls durch die Effizienz im Umgang mit den Betriebsmitteln aus. Diese beschreibt das Verhältnis zwischen den benötigten Betriebsmitteln und dem erreichtem Leistungsniveau. Die Teilaspekte, die zu betrachten sind, um die Qualität hinsichtlich der Effizienz bewerten zu können, sind Zeit- und Verbrauchsverhalten.

Software, die erfolgreich ist, wird immer weiterentwickelt. In ihrem Lebenszyklus treten deswegen ständig neue Anforderungen auf, welche Änderungen an der Software nach sich ziehen. Software ist aber auch in vielen Fällen fehleranfällig, weshalb im Zuge der Fehlerbeseitigung ebenfalls Änderungen an der Software nötig sind. Der erforderliche Aufwand um Änderungen vorzunehmen bestimmt maßgeblich die Qualität von Software. Die zu bewertenden Merkmale sind hierbei Analysierbarkeit, Stabilität, Modifizierbarkeit und Testbarkeit. Das letzte Merkmal bzw. dessen Umsetzung spielt eine Kernrolle in Kapitel 4.2.

Da sich die IT-Landschaft rasend schnell weiterentwickelt, gibt es eine immer größere Anzahl von verschiedenen Umgebungen in der Software betrieben werden kann. Diese Umgebungen können zum Beispiel ein Web-Server, eine Betriebssystem oder ein komplett neues Gerät sein. Der Boom von Smartphones und den damit verbundenen Apps ist ein Paradebeispiel hierfür.

Die Übertragbarkeit gibt den Grad der Leichtigkeit an, mit welcher Software in eine neue Umgebung portiert werden kann und wird weiterhin durch Anpassbarkeit, Installierbarkeit, Koexistenz und Austauschbarkeit bestimmt.

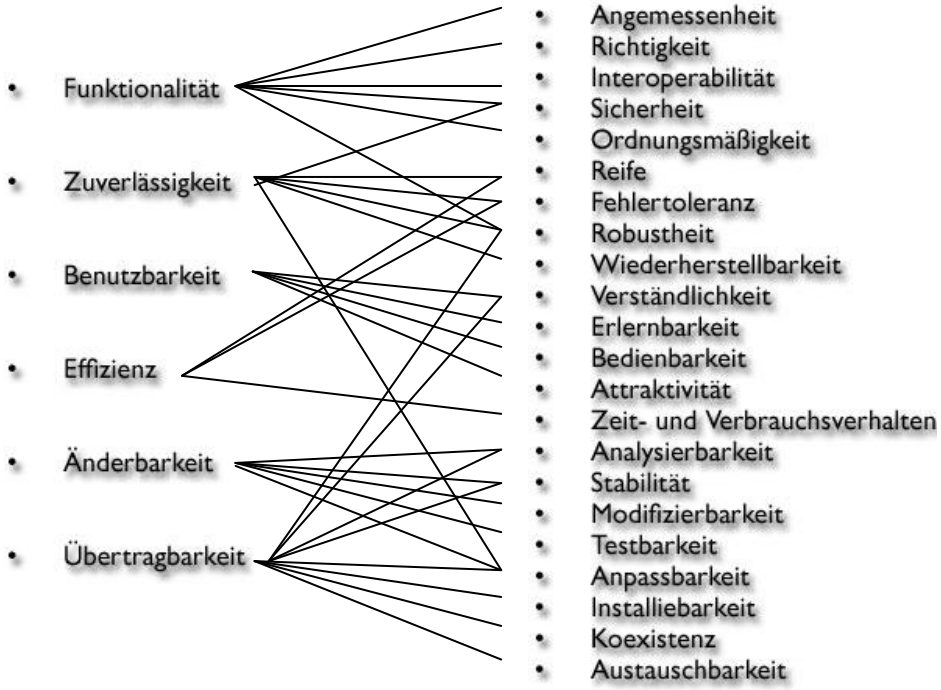


Abbildung 2–1: Zerlegung und Abhängigkeiten der Qualitätsmerkmale

2.1.2 Qualitätsmerkmale speziell für Webapplikation

Die in Abschnitt 2.1.1 genannten Qualitätsmerkmale sind allgemeingültig für jeden Typ von Software unabhängig davon ob diese im Mainframe-, Client-/Server- oder Web-Umfeld betrieben wird. Für Software aus dem Web-Umfeld treten aber noch weitergehende Anforderungen in Hinblick auf Qualität auf, die erfüllt werden müssen. Es handelt sich hierbei um Qualitätsmerkmale, welche der Benutzbarkeit zuzuordnen sind.

Da Software im Web-Umfeld nicht direkt bei dem Benutzer installiert wird, sondern in Form einer Webseite oder eines Webservice lediglich abgerufen wird, ist es essentiell, dass das Angebot leicht aufzufinden ist.

Dazu zählen zum einen eine sprechende und leicht zu merkende Web-Adresse und zum anderen die Anzeige der Webseite in den Toptreffern von Suchmaschinen.

Barrierefreiheit ist ein weiteres besonderes Qualitätsmerkmal von Web-Anwendungen. Diese ermöglicht es auch behinderten Personen ein Web-Angebot zu nutzen und ist im Allgemeinen notwendig um standardkonforme Web-Angebote bereit zu stellen.

Barrierefreiheit beeinflusst somit nicht unerheblich die Auffindbarkeit. Inwiefern ein Web-Angebot durch Suchmaschinen gefunden wird, hängt auch davon ab, ob das Web-Angebot barrierefrei bzw. standardkonform ist.

Nach §4 des Behindertengleichstellungsgesetzes ist Barrierefreiheit folgendermaßen definiert (vgl. [URL:BGG]):

„Barrierefrei sind bauliche und sonstige Anlagen, Verkehrsmittel, technische Gebrauchsgegenstände, Systeme der Informationsverarbeitung, akustische und visuelle Informationsquellen und Kommunikationseinrichtungen sowie andere gestaltete Lebensbereiche, wenn sie für behinderte Menschen in der allgemein üblichen Wiese, ohne besondere Erschwernis und grundsätzlich ohne fremde Hilfe zugänglich und nutzbar sind."

Die genauen Anforderungen an ein Webangebot werden in der Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz definiert. (vgl. [URL:BITV])

Da Web-Angebote weltweit von jedem aufrufbar sind, liegt ein besonderes Augenmerk auf der Rechtskonfomität.

Es dürfen zum Beispiel nicht die Urheberrechte von Grafiken und Texten verletzt werden oder Links veröffentlicht werden, die zu Seiten mit illegalem Inhalt führen. Im Falle eines kommerziellen Web-Angebots, zum Beispiel in Form eines Web-Shops, muss diesem Qualitätsmerkmal eine noch größere Bedeutung beigemessen werden, da Fehler in diesem Bereich fatal sein können.

2.2 Testmodelle

„Um eine strukturierte und steuerbare Softwareentwicklung durchzuführen, werden Softwareentwicklungsmodelle […] eingesetzt." (vgl. [Spi05], S. 18)

Es gibt eine Vielzahl davon und sowohl das V-Modell als auch Extreme Programming sollen im Folgenden kurz angerissen werden, da diese den Prozess des Testens integrieren. Diese Modelle genügen allerdings nicht um den Testprozess selbst zu planen, sondern ordnen diesen lediglich in den Projektablauf ein. Das Management des Testens wird in Kapitel 5.4 näher beschrieben.

2.2.1 Verifikations-Modell

Das Verifikations- oder kurz V-Modell nach Boehm (vgl. Abbildung 2–2) trennt den Entwicklungs- vom Testprozess und betrachtet beide als gleichwertig. Die Phasen der beiden Prozesse stehen sich dabei gegenüber und sollten möglichst zeitnah zueinander, realisiert werden (vgl. [Wal90], S. 85). Dabei bildet der linke Ast die Entwicklungsschritte von der Planung bis zur programmiertechnischen Umsetzung eines Projektes ab. Dieser Ast entspringt dem klassischen Wasserfall-Modell. Der rechte Ast skizziert die Testschritte von einzelnen Komponenten- oder Unit-Tests über den Integrations-, den Akzeptanz-/System-, den Performanz-/Lasttest bis hin zur Inbetriebnahme.

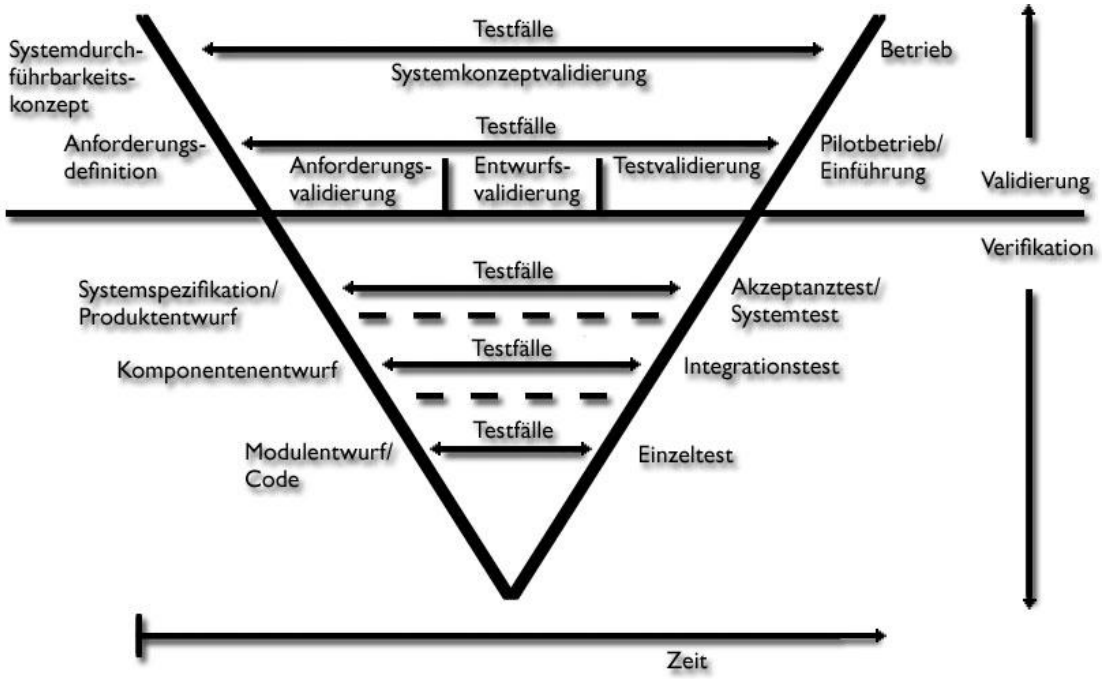


Abbildung 2–2: V-Modell nach Boehm (vgl. [Boe79])

Je kürzer ein Fehler im System verbleibt desto weniger zusätzliche Kosten und Aufwand bedeutet dies. Auf Grund der zeitnahen Validierung und Verifikation der Entwicklungsergebnisse durch eine korrespondierende Testphase auf gleicher Abstraktionsebene, wird die Verweildauer von Fehlern in der Software erheblich verkürzt und unübersehbare Folgen des Fehlers werden minimiert. Durch das V-Modell steht somit früher ein konsistentes System zur Verfügung, dessen Reife relativ genau definiert werden kann, vorausgesetzt die Testphase hat einen ausreichenden und aussagekräftigen Umfang. Die einzelnen Testphasen werden in Kapitel 5 näher betrachtet.

2.2.2 Extreme Programming

Extreme Programming oder kurz XP ist eine Methode der Agilen Softwareentwicklung, welche lösungsorientiert ist und dabei ein formalisiertes Vorgehen als niederprior ansieht. Dieses Vorgehensmodell (vgl. Abbildung 2–3) ist besonders kundennah und liefert auf Grund der kurzen Entwicklungszyklen umgehend nutzbare Software. Dabei wird jeder Entwicklungszyklus mit einer sogenannten User-Story begonnen und endet mit Software, welche neue Funktionalitäten aufweist und vom Kunden bereits gesichtet werden kann. Während den Entwicklungszyklen liegt ein besonderes Augenmerk auf testgetriebener Entwicklung, auf welche in Kapitel 4.2 näher eingegangen wird.

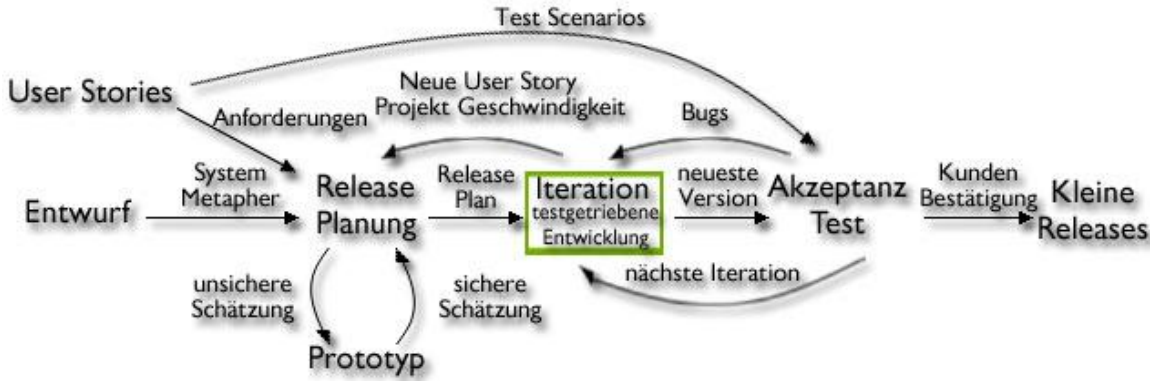


Abbildung 2–3: XP Projekt (vgl. [URL:XP])

Die Basis für XP ist eine gute Kommunikation zwischen Kunde und Entwickler, welche aussagekräftige Feedback-Mechanismen benötigt. Eine Beispiel hierfür sind die Tests, die begleitend zur Entwicklung ab Tag 1 geschrieben, gepflegt und kontinuierlich integriert werden. Durch die Idee des paarweisen Programmierens ist allerdings auch unterhalb der Entwickler eine solide und respektvolle Kommunikation notwendig. Ein großer Vorteil des paarweisen Programmierens sind seltener entstehende Wissensmonopole, wodurch wiederum Abhängigkeiten von einzelnen Projektbeteiligten reduziert werden. *„Es ist das Ziel, den Entwicklungsprozess an die örtlichen Begebenheiten anzupassen und fortlaufend zu verbessern.“* (vgl. [URL:WEST])

Der bereits erwähnte Ansatz der testgetriebenen Entwicklung unterstützt die Forderung nach Einfachheit der Software und liefert diese nach Bedarf, das heißt es werden genau die Anforderungen und nicht mehr erfüllt. Dadurch entsteht Software, welche sauberen, flexiblen, mitunter fehlerfreieren und einfachen Code aufweist. Eine Folge hiervon ist ein System, das so früh wie möglich ausgeliefert werden kann.

2.2.3 Grundlegende Testarten

In den Stufen der Testmodelle gibt es verschiedene Anforderungen an Tests hinsichtlich was und wie getestet wird. Um einen exakten Testplan aufstellen zu können, muss dementsprechend differenziert werden, welche Arten von Tests wann zum Einsatz kommen.

Es wird hier zwischen 4 grundlegenden Testarten unterschieden: (vgl. [Spi05], S. 69)

- funktionaler Test
- nicht-funktionaler Test
- strukturbezogener Test
- änderungsbezogener Test

Für diese Arbeit sind der funktionale, nicht-funktionale und änderungsbezogene Test relevant. Die Grundzüge dieser 3 Testarten sollen in diesem Kapitel näher erläutert werden.

2.2.3.1 Funktionaler Test

Wie sich aus dem Namen ableiten lässt, handelt es sich bei dieser Testart um *„alle Testmethoden, mittels derer das von außen sichtbare Ein- und Ausgabeverhalten eines Testobjekts geprüft wird.“* (vgl. [Spi05], S. 69) Zu diesen Funktionalitäten zählt z.B. die Ausgabe eines Browsers nach Abschicken eines Formulars. Aus diesem Grund kommen die in Abschnitt 2.3.1 betrachteten Blackbox-Verfahren zur Testfallermittlung zum Einsatz.

Die Grundlagen dieser Testart bilden somit die gestellten Anforderungen an das Testobjekt, welche die Funktionalität und das Verhalten exakt definieren. Nach [ISO 9126] sind die grundlegenden Merkmale von Funktionalität: Richtigkeit, Interoperabilität, Sicherheit, Angemessenheit und Ordnungsmäßigkeit.

Meistens liegen solche Anforderungen in einem Lasten- oder auch Pflichtenheft vor und geben dem System bzw. Testobjekt einen präzisen Rahmen was es können muss. Allerdings können Anforderungen auch nur grob oder vage vorliegen. Im Zuge, der in Kapitel 4.2 betrachteten, testgetriebenen Entwicklung werden die Anforderungen in solch einem Fall erst während der Entwicklungsarbeit spezifiziert. Laut (vgl. [Spi05], S. 70) ist folgendes entscheidend: *„Sind die einmal definierten Testfälle (…) fehlerfrei gelaufen, wird die entsprechende Funktionalität als validiert betrachtet.“*

Für Testfälle, die erneut laufen müssen, kommen änderungsbezogene Tests, welche auch Regressionstests genannt werden, zum Einsatz.

Bei funktionalen Tests wird weiterhin zwischen anforderungsbasierten Tests und geschäftsprozessorientierten Tests differenziert.

Anforderungsbasierte Tests finden meist in den niedrigeren Teststufen, wie dem Komponententest Anwendung, da nur einzelne Funktionalitäten geprüft werden, wie z.B. das Abschicken von Rechnungen..

Geschäftsprozessbasierte Tests hingegen werden bevorzugt in Teststufen wie dem Last- und Performanztest oder auch Akzeptanztest verwendet, da so ganze Abläufe von realistischen Nutzeraktionen, z.B. Transaktionen mit Kunden auf einem Webportal, simuliert und getestet werden können.

Nach [ISO 9126] gehören Kriterien wie Benutzbarkeit oder Effizienz ebenfalls zum Begriff der Qualität von Software. Diese stehen nur indirekt mit der Funktionalität von Software in Verbindung, müssen aber ebenfalls den Anforderungen an das Testobjekt genügen und getestet werden. Eine Betrachtung dessen folgt im nächsten Abschnitt.

2.2.3.2 Nichtfunktionaler Test

Die nicht-funktionalen Anforderungen bestimmen maßgeblich die Qualität von Software und somit auch die Zufriedenheit des Kunden. Diese sind zwar meistens nicht klar definiert wie z.B. eine hohe Usability, womit sie auch schwer oder mitunter nicht zu testen sind. Aber Software mit voller Funktionalität, welche der Endanwender allerdings nicht oder nur schwer nutzen kann, erbringt keinen großen Nutzen. Aus diesem Grund müssen nicht-funktionale Anforderungen, die nicht klar definiert sind, präzisiert werden, um sie testen zu können. Ein Beispiel hierfür ist die durchschnittliche Antwortzeit eines Webservers in Millisekunden.

Nach (vgl. [Spi05]) kommen nicht-funktionale Tests hauptsächlich in der Teststufe Systemtest zum Einsatz und sollten nach (vgl. [Mey82]) und (vgl. [Wal90]) folgende Testtypen abdecken:

- Lasttest
- Performanztest
- Volumen-/Massetest
- Stresstest
- Sicherheitstest
- Stabilitätstest

- Robustheitstest
- Kompatibilitätstest
- Test unterschiedlicher Konfigurationen
- Test der Benutzerfreundlichkeit
- Prüfung der Dokumentation
- Prüfung der Änderbarkeit und Wartbarkeit

Gemäß [ISO 9126] gehören zu den Qualitäts-Kriterien von Software: Zuverlässigkeit, Benutzbarkeit und Effizienz. Darüber hinaus zählen Änderbarkeit und Übertragbarkeit auch dazu, da Wartungsarbeiten und ähnliches nicht das Nutzerverhalten bzw. die Qualität der Software beeinflussen sollten.

Änderbarkeit und Übertragbarkeit werden durch die strukturelle Qualität von Software bestimmt und tragen maßgeblich zur Gesamtqualität bei. Laut (vgl. [Wes06], S. 4) wird die Qualität von Software durch die minimale Schnittmenge der erfüllten funktionalen und nicht-funktionalen Anforderungen bestimmt. Das bedeutet, Software, welche die funktionalen Anforderungen erfüllt aber nicht die nicht-funktionalen, hat keinen Wert.

Da sich Software also ändern bzw. weiterentwickeln kann, auf ein anderes System übertragen werden soll oder gewartet wird, müssen die bereits erstellten Tests wiederholt ausgeführt werden. Diese Tests werden änderungsbezogene Tests oder auch Regressionstests genannt.

2.2.3.3 Änderungsbezogener Test

Laut (vgl. [Spi05], S. 74) ist ein änderungsbezogener Test

„ein erneuter Test eines bereits getesteten Programms nach dessen Modifikation mit dem Ziel, nachzuweisen, dass durch die vorgenommenen Änderungen keine neuen Defekte eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden.“

Je nach Komplexität des Projekts können die Auswirkungen von Änderungen auf andere Software- oder Systemteile nicht mehr zuverlässig eingeschätzt werden. Dies trifft noch mehr für Projekte zu, die schlecht dokumentiert sind. Dementsprechend kann es notwendig sein, nicht nur die Tests der geänderten oder neuen Systemteile auszuführen, sondern die gesamte Testsuite. Wenn die gesamte Testsuite auf Grund von Zeitproblemen nicht ausgeführt werden kann, empfiehlt es sich, eine Wiederholungstestmatrix (vgl. Abbildung 2–

4) aufzustellen, um einen präzisen Überblick der auszuführenden Testfälle zu erhalten. (vgl. [Het93])

Im Falle eines Fehlernachtests kann es mitunter genügen, die Testfälle erneut auszuführen, in denen vorher Fehler aufgedeckt wurden. Sollte aber die gesamte Systemlandschaft geändert worden sein und nicht nur Teile der Software selbst, ist ein vollständiger Regressionstest unerlässlich. Das bedeutet, dass die Ergebnisse aller vorher gelaufenen Tests erneut verifiziert werden müssen.

Die Referenz in Form von Sollwerten für folgende Tests bilden somit die Ergebnisse vorangegangener Tests (vgl. [Tra96], S. 234). Die Tests müssen aus diesem Grund ausreichend dokumentiert sein, was in diesem Zusammenhang eine Wiederholbarkeit sicherstellt. Diese können sich in allen Teststufen befinden und sowohl funktionale als auch nicht-funktionale Tests umfassen. (vgl. [Spi05], S. 74)

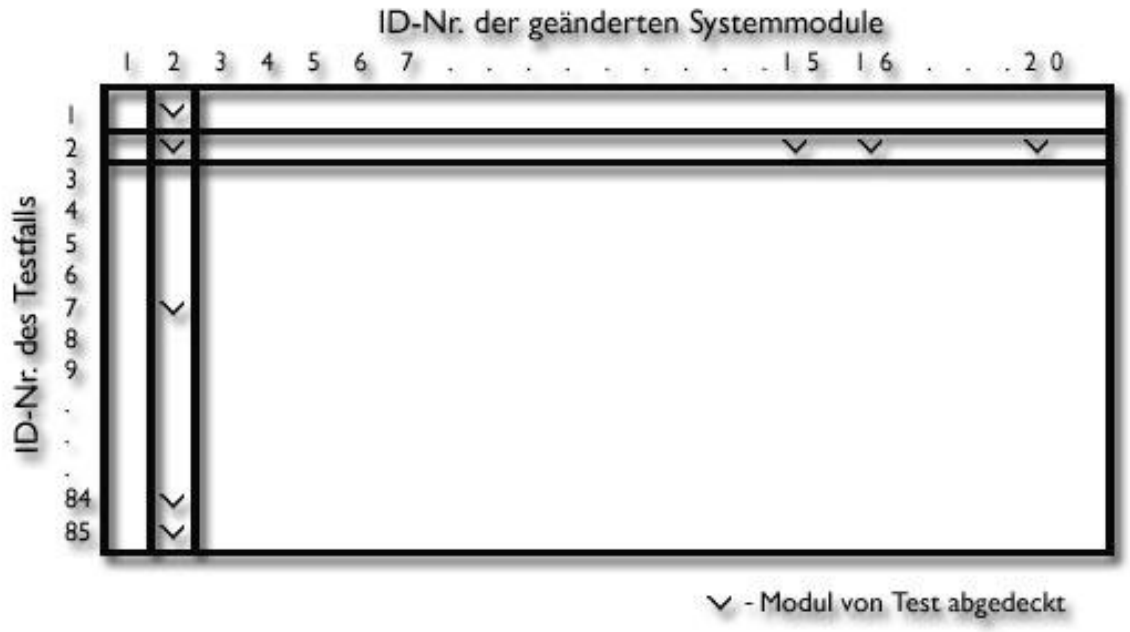


Abbildung 2–4: Wiederholungstestmatrix (vgl. [Het93])

Eingeschlichene Folgefehler erst im Nachhinein zu beheben, wenn sie im Betrieb auftreten, führt zu unnötig erhöhten Kosten und Aufwand. Daher ist es in einem Testzyklus essentiell, dass Software nicht nur fehlerfrei ist, sondern auch bleibt und dass dieser Zustand nachgewiesen werden kann.

2.3 Dynamischer Test

Neben statischen Tests wie Code-Walkthrough, Review oder Kontrollflussanalyse haben dynamische Tests eine besondere Bedeutung für diese Arbeit. Dynamische Tests unterscheiden sich von statischen Tests dahingehend, dass das Testobjekt auf einem Computer zur Ausführung gebracht und nicht nur von außen betrachtet wird.

Diese werden in Blackbox- und Whitebox-Verfahren unterteilt und dienen der systematischen Spezifikation von Testfällen. Beide Testfallentwurfsverfahren sollen im Folgenden näher betrachtet werden.

2.3.1 Blackbox-Verfahren

Bei dem Blackbox-Verfahren wird die Software als schwarzer Kasten angesehen, wobei laut (vgl. [Spi05], S. 108 ff) keinerlei Kenntnisse über den Programmcode und inneren Aufbau vorhanden sind. Dies ist auch nicht notwendig, da das Testobjekt von außen betrachtet wird (PoO – Point of Observation liegt außerhalb) und nur durch Eingabedaten gesteuert werden kann (PoC – Point of Control liegt außerhalb). Somit werden die Testfälle ausschließlich über vorhandene Anforderungen spezifiziert. (vgl. Abbildung 2–5)

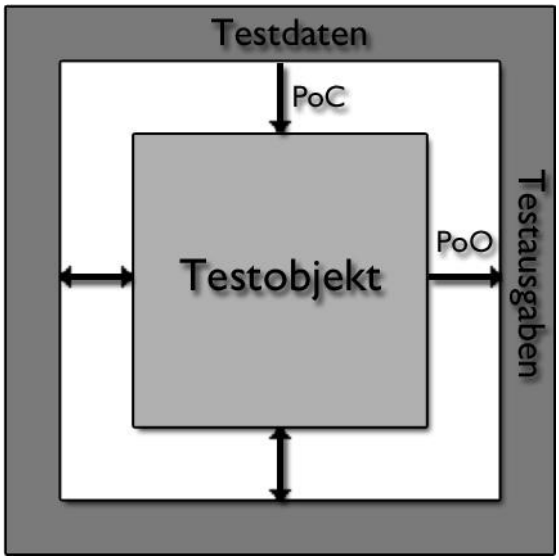


Abbildung 2–5: PoO und PoC bei Blackbox-Verfahren außerhalb (vgl. [Spi05], S. 108)

Aus diesem Grund eignen sich Blackbox-Verfahren in höheren Teststufen wie dem Leistungs- oder Akzeptanztest zur Testfallspezifikation. Ansätze, wie die in Kapitel 4.2 beschriebene testgetriebene Entwicklung, bedienen sich der Blackbox-Verfahren allerdings auch schon in der Stufe des Komponententests.

Da die Spezifikation der Testfälle ausschließlich über die Anforderungen an das Testobjekt geschieht, werden nur die möglichen Eingabedaten herangezogen. Somit kann die Anzahl der Testfälle auf Grund der vielfältigen Möglichkeiten von Eingabedaten zu groß ausfallen, um in einem angemessenen Rahmen realisiert zu werden und es muss eine sinnvolle Auswahl getroffen werden. Unter den Blackbox-Verfahren finden sich verschiedene Methoden, die sich dafür anbieten und im Folgenden kurz betrachtet werden.

2.3.1.1 Äquivalenzklassenbildung

Nach (vgl. [Fra07], S.32) wird bei der Äquivalenzklassenbildung *„die Menge der möglichen Testfälle anhand der in den Anforderungsspezifikationen beschriebenen Bedingungen in einen endliche Zahl von äquivalenten Klassen unterteilt."* Es werden also alle möglichen Eingabedaten in Äquivalenzklassen unterteilt. Dabei wird für jeden Repräsentanten einer Klasse davon ausgegangen, dass die Wirkung auf das Testobjekt die gleiche ist (vgl. [Wal90], S. 176). Es ist daher für die Spezifikation eines Testfalls ausreichend lediglich einen Repräsentanten pro Äquivalenzklasse aus der Menge von Eingabewerten auszuwählen.

Dabei ist zu beachten, dass immer eine gültige und eine entsprechende ungültige Äquivalenzklasse zu bilden ist. Die gültige enthält hierbei die Eingabewerte, die innerhalb des definierten Wertebereichs liegen, und die ungültige jene, die außerhalb liegen. Anschließend wird aus jeder gültigen und ungültigen Klasse ein Repräsentant für den konkreten Testfall ausgewählt. (vgl. Abbildung 2–6)

Ein-/Ausgabegröße	Äquivalenzklasse			
	gültige		ungültige	
Tagesdatum	≥ 1, TF1	≤ 31, TF2	< 1, TF3	>31, TF4

Konkrete Testfälle: TF1: 25

TF3: 0

T4: 40

Abbildung 2–6: Hilfsschema für die Äquivalenzklassenbildung (vgl. [Wal90], S. 176)

Nach (vgl. [Spi05], S. 121) bietet sich die Äquivalenzklassenbildung für den System-, Integrations- und Komponententest an.

2.3.1.2 Grenzwertanalyse

„Die Grenzwertanalyse stellt sicher, dass Fehler in den kritischen Grenzbereichen der Äquivalenzklassen gefunden werden“ (vgl. [Fra07], S. 36) und bildet somit eine sinnvolle Ergänzung zur Äquivalenzklassenbildung. Besonders in den Randbereichen werden in Programmen die eigentlichen Fallunterscheidungen getroffen, weshalb die Fehleranfälligkeit in diesen Bereichen besonders hoch ist.

Es werden somit die bereits erstellten Äquivalenzklassen herangezogen. Aber in diesem Fall werden die Repräsentanten für die Testfälle nicht willkürlich sondern bewusst aus den Randbereichen gewählt. (vgl. Abbildung 2–7)

Äquivalenzklasse	Grenze	Grenzwerte	TF Nr.	Steuersatz
Einkommenssteuer	80‘000 .-	79‘999 .-	TF1	20%
		80‘000 .-	TF2	25%
		80‘001 .-	TF3	25%
	110‘000 .-	109‘999 .-	TF4	25%
		110‘000 .-	TF5	25%
		110‘001 .-	TF6	30%

Abbildung 2–7: Beispiel für Grenzwertanalyse (vgl. [Wal90], S. 177)

Laut (vgl. [Spi05], S. 121) werden *„an jedem Rand […] der exakte Grenzwert und die beiden (innerhalb und außerhalb der Äquivalenzklasse) benachbarten Werte getestet.“* Überschneiden sich hierbei Testfälle von Äquivalenzklassen werden diese zusammengefasst.

Es gibt noch weitere Analyseverfahren, wie die Ursache-Wirkungs-Graph-Analyse oder der zustandsbezogene Test aber diese sind für die Arbeit nicht relevant.

2.3.2 Whitebox-Verfahren

Im Gegensatz zu den Blackbox-Verfahren wird die Software im Falle von Whitebox-Verfahren als weißer und somit einsehbarer Kasten betrachtet. Weiterhin bieten sich Whitebox-Verfahren eher für die unteren Teststufen wie den Komponententest an und sind zum Beispiel für den Systemtest wenig sinnvoll. (vgl. [Spi05], S. 109)

Das bedeutet, dass Programminterna hinsichtlich der Erfüllung von Spezifikationen analysiert werden. Informationen zu den Analysen bzw. der Logik und inneren Struktur der Software können beispielsweise direkt aus dem Quellcode durch ein Review extrahiert oder aus Kontroll- und Datenflussdiagrammen gewonnen werden. (PoO liegt innerhalb des Testobjekts, vgl. Abbildung 2–8) Es muss entschieden werden, welche Anweisungen und Pfade der Software in Tests ausgeführt werden, um die gewünschten Funktionalitäten zu erreichen. Diese Analysen betreffen somit zum einen die Testüberdeckung, welche prüft, ob alle (essentiellen) Programmteile durchlaufen wurden, oder manche Programmteile gar nicht erreichbar sind und zum anderen die Strukturkomplexität. (vgl. [Wal90], S. 179)

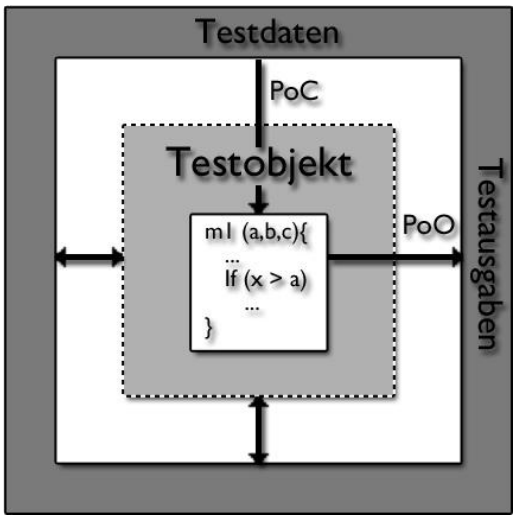


Abbildung 2–8: PoO und PoC bei Whitebox-Verfahren innerhalb (vgl. [Spi05], S. 108)

Besonders bei komplexen Projekten ist ein vollständiger Whitebox-Test und somit eine vollständige Testüberdeckung, was das Ausführen von allen Programmpfaden bedeutet, unrealistisch. Da fehlende Pfade oder Spezifikationen bei dieser Testart nicht gefunden werden können, wäre ohnehin keine Fehlerfreiheit des Testobjekts garantiert. Somit ist das

Erreichen von vorher definierten Zielwerten für die verschiedenen Testüberdeckungskenngrößen ausreichend. (vgl. [Wal90], S. 180ff) Der Nachweis der erreichten Testüberdeckungskenngrößen kann nicht manuell erfolgen, sondern muss durch sogenannte Code Coverage Tools unterstützt werden. (vgl. Kapitel 4.4) Die Kenngrößen werden im Folgenden näher betrachtet.

2.3.2.1 Anweisungsüberdeckung

Laut (vgl. [Wal90], S. 180) wird unter Anweisungsüberdeckung, Statement Coverage oder auch dem C0-Maß *„das Verhältnis der Anzahl der durchlaufenen Anweisungen zur Gesamtzahl der Anweisungen eines Testobjekt“* verstanden. Dafür bietet es sich an, den Programmcode in ein Kontrollflussdiagramm zu übertragen (vgl. Abbildung 2–9), um die geforderten Überdeckungen genauer zu spezifizieren. (vgl. [Spi05], S.144) Im gegebenen Beispiel wäre eine volle Anweisungsüberdeckung zu erreichen, indem die Kanten a, b, f, g, h, d, e in einem Testfall durchlaufen werden.

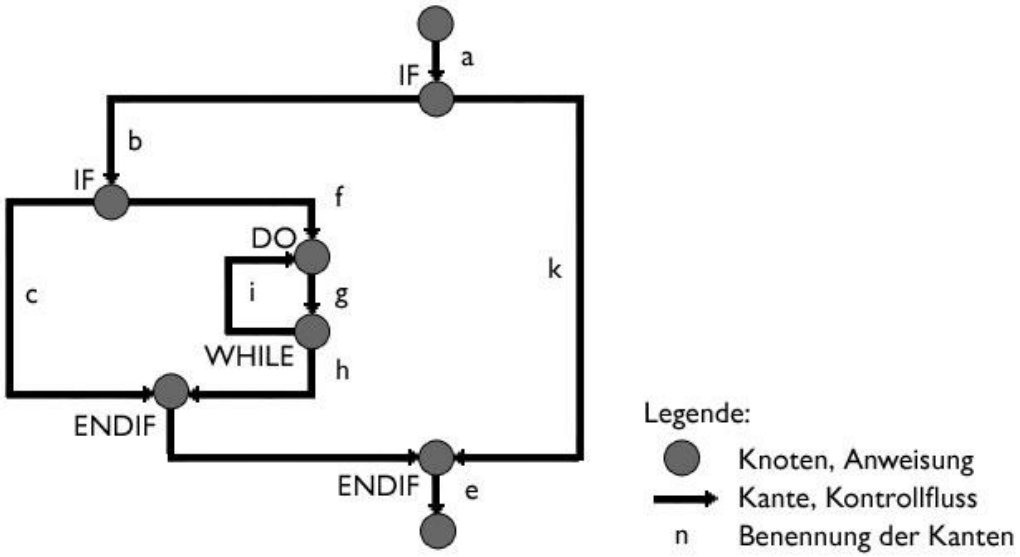


Abbildung 2–9: Kontrolflussdiagramm (vgl. [Spi05], S. 145)

Laut (vgl. [Wal90], S. 180) ist 95% ein in der Praxis typischer Zielwert für die Anweisungsüberdeckung, d.h. 95% aller Anweisungen einer Software müssen in Testfällen zur Ausführung gebracht werden.

Es kann zwei Gründe geben warum ein Teil des Codes nicht durchlaufen wurde. Entweder wurde einfach kein Testfall spezifiziert, um die jeweilige Anweisung zu durchlaufen. In diesem Fall müssen die Testfälle angepasst bzw. erweitert werden. Oder der Quell-Code ist

fehlerhaft und lässt das Erreichen der Anweisung nicht zu, womit der Quell-Code angepasst werden muss.

2.3.2.2 Zweigüberdeckung

Laut (vgl. [Wal90], S. 180) wird unter Zweigüberdeckung, Decision and Branch Coverage oder auch dem C1-Maß *„das Verhältnis von durchlaufenen Zweigen zu allen möglichen Zweigen des Testobjekt."* verstanden. Nach (vgl. [Spi05], S. 147) ist die Zweigüberdeckung ein weitergehendes und umfassenderes Kriterium, da es nicht ausreicht, lediglich jede Anweisung auszuführen. Im Kontrollflussdiagramm (vgl. Abbildung 2–9) muss zur vollständigen Zweigüberdeckung jede Kante durchlaufen, d.h. jede Bedingung ausgewertet werden.

„Die Zweigüberdeckung verlangt, dass bei einer Verzweigung des Kontrollflusses (bei einer Bedingung) beide bzw. (bei einer CASE-Anweisung) alle Möglichkeiten und bei Schleifen neben dem Schleifenkörper auch die Umgehung bzw. ein Rücksprung zum Schleifenanfang zu berücksichtigen sind." (vgl. [Spi05], S. 147)

Für obiges Beispiel (vgl. Abbildung 2–9) wären 3 Testfälle, die folgende Kanten durchlaufen, zu definieren:

- a, b, c, d, e
- a, b, f, g, i, g, g, h, d, e
- a, k, e

In der Objektorientierten Programmierung ist sowohl die Anweisungs- als auch die Zweigüberdeckung unzureichend, da der Kontrollfluss in Klassen eine eher geringere Komplexität aufweist, ganz im Gegensatz zu den Beziehungen der Klassen untereinander. Hierfür werden weitere Testüberdeckungskenngrößen, wie Attribute Coverage (jedes Objektattribut wird mindestens einmal modifiziert), Interface Coverage (jede Nachricht wird mindestens einmal gesendet) oder Parameter Coverage (jeder Parameter jeder Nachricht wird mindestens einmal modifiziert) benötigt, die sich nicht auf den Kontrollfluss gründen. Allerdings finden diese Überdeckungskenngrößen noch geringe Anwendung in der Praxis. (vgl. [Fra07], S. 56) Die Anweisungs- und Zweigüberdeckung bieten aber gute Möglichkeiten, nicht aufgerufene Methoden in Tests zu identifizieren. (vgl. [Spi05], S. 149)

Laut (vgl. [Wal90], S. 180) ist 85% ein in der Praxis typischer Zielwert für die Zweigüberdeckung, d.h. 85% aller Zweige einer Software müssen in Testfällen durchlaufen werden.

2.3.2.3 Bedingungsüberdeckung

Laut (vgl. [Wal90], S. 180) wird unter einfacher Bedingungsüberdeckung, Branch Coverage oder auch dem C2-Maß das gleiche verstanden

„wie C1, aber statt Zweige werden Terme innerhalb von Ausdrücken verwendet. Eine Testüberdeckung von 100% C2 bedeutet, dass in einem Programmabschnitt innerhalb jedes Ausdrucks einer Bedingungsanweisung oder Schleifenanweisung jeder Term mindestens einmal evaluiert wurde.“

Somit muss jede Bedingung oder Schleife den Ergebniswahrheitswert „wahr“ und „falsch“ in den Testfällen annehmen. Da Bedingungen durch logische Operatoren miteinander verbunden sein können, muss diese unterschiedliche Komplexität berücksichtigt werden. (vgl. [Spi05], S. 149)

Der einfachen Bedingungsüberdeckung genügt es, dass jede atomare Teilbedingung den Wert „wahr“ und „falsch“ in Testfällen annehmen muss.

„Eine atomare Teilbedingung ist eine Bedingung, die keine logischen Operatoren […] sondern höchstens Relationssysmbole wie ‚>‘ oder ‚=‘ enthält. Eine Bedingung im Programmtext des Testobjekts kann aus mehreren atomaren Teilbedingungen zusammengesetzt sein.“ (vgl. [Spi05], S. 150)

Ein Beispiel soll die einfache Bedingungsüberdeckung veranschaulichen. Laut der obigen Definitionen würde die Bedingung „x > 10 OR y < 100“ aus zwei atomaren Teilbedingungen bestehen, nämlich „x > 10“ und „y < 100“. Um eine einfache Bedingungsüberdeckung zu erreichen, könnte ein Testfall mit folgenden Testdaten ausgeführt werden:

- x=11 und y=101
- x=9 und y=99

Im ersten Fall führt dies bei „x > 10“ zu wahr und „y < 100“ zu falsch. Auf Grund der Verknüpfung ist die Gesamtbedingung „wahr“. Im zweiten Testfall lautet das Ergebnis für „x > 10“ falsch und für „y < 100“ wahr, womit die einfache Bedingungsüberdeckung erreicht wäre, da jede atomare Teilbedingung den Wert „wahr“ und „falsch“ angenommen hat.

Allerdings ist die Gesamtbedingung in beiden Fällen auf Grund der logischen Verknüpfung „wahr“, womit die einfache Bedingungsüberdeckung ein schwächeres Kriterium als die Anweisungs- oder Zweigüberdeckung ist, da dieser Umstand nicht berücksichtigt werden muss. (vgl. [Spi05], S. 150)

Wenn dies in Testfällen beachtet werden soll, muss die Mehrfachbedingungsüberdeckung, Branch Condition Coverage oder auch das C3-Maß herangezogen werden.

„Um eine Testüberdeckung von 100% C3 zu erhalten, müssen alle möglichen Kombinationen von Elementarbedingungen innerhalb einer Abfrage oder Schleifenbedingung einmal durchlaufen werden.“ (vgl. [Wal90], S. 180)

Für das oben ausgeführte Beispiel würden sich in diesem Fall folgende Testdaten ergeben:

- x=11 und y=99, (x > 10 (W) OR y < 100 (W)) (W)
- x=11 und y=101, (x > 10 (W) OR y < 100 (F)) (W)
- x=9 und y=99, (x > 10 (F) OR y < 100 (W)) (W)
- x=9 und y=101, (x > 10 (F) OR y < 100 (F)) (F)

Für diese Testdaten nehmen alle atomaren Teilbedingungen und nun auch die Gesamtbedingung beide Wahrheitswerte an. Nach (vgl. [Spi05], S. 150) erfüllt die Mehrfachbedingungsüberdeckung

„somit auch die Kriterien der Anweisungs- und Zweigüberdeckung. Sie ist ein umfassenderes Kriterium […]. Allerdings ist sie auch sehr aufwändig, da bei steigender Anzahl der atomaren Bedingungen die Zahl der möglichen Kombinationen exponentiell ansteigt.“

In der Praxis kann es vorkommen, dass nicht alle Kombinationen der atomaren Teilbedingungen mit Testdaten realisiert werden können. In solch einem Fall ist eine minimale Mehrfachbedingungsüberdeckung ausreichend. Hierfür werden nur die atomaren Teilbedingungen berücksichtigt, welche die Gesamtbedingung beeinflussen.

Wenn die Komplexität der Bedingungen für Testfälle von Bedeutung ist, sollte eine vollständige Prüfung und somit 100% Überdeckung erreicht werden. Falls nicht, ist Zweigüberdeckung ausreichend. (vgl. [Spi05], S. 153)

Neben den bisher beschriebenen Testüberdeckungskenngrößen gibt es noch eine Vielzahl anderer, die aber auf Grund des zu hohen Testaufwands nicht praktikabel oder einfach nicht mehr zeitgemäß sind und hier nur kurz genannt werden sollen. Dazu zählen unter anderem

die Pfadüberdeckung (Path Coverage oder auch C4-Maß), welche besonders in der objektorientierten Programmierung unzureichend und veraltet ist oder die Schleifenüberdeckung (Loop Coverage).

2.4 Zusammenfassung

Das zweite Kapitel hat bisher die Grundlagen des Testens vermittelt. Dazu wurden zunächst die Qualitätsmerkmale beschrieben, die Softwareprojekte aufweisen sollten und wie diese erreicht werden. Desweiteren wurden verschiedene Modelle betrachtet, um den Testprozess in den Softwareentwicklungsprozess einzugliedern und die grundlegenden Arten von Tests erläutert. Eine Analyse der Methoden, um Testfälle in der Praxis zu erstellen, bildet den Schlusspunkt. Das nächste Kapitel geht auf die Anforderungsanalyse der Arbeit ein.

3 Anforderungsanalyse und Zielstellung

In diesem Kapitel wird das Projekt, mit welchem sich diese Diplomarbeit beschäftigt, näher betrachtet. Weiterhin werden die Ziele und daraus resultierenden Anforderungen analysiert.

3.1 Hoga Jobportal

Das Projektumfeld, in welchem sich diese Diplomarbeit bewegt, ist das Hoga Jobportal. (vgl. Abbildung 3–1) Dies ist ein komplexes Webportal auf Basis des WCMS TYPO3. Es bietet die Möglichkeit, Arbeitssuchenden ein Stellengesuch im Bereich der Hotel und Gastronomie zu schalten oder direkt nach Stellenangeboten zu suchen. Arbeitgeber können ebendiese Stellenangebote schalten und auf Stellengesuche antworten. Darüber hinaus kann der gesamte Bewerbungsprozess über das Webportal durchgeführt werden. Dabei können Stellenangebote und –gesuche im gesamten deutschsprachigen Raum geschalten werden.

Für die Medienkombinat GmbH stellt das Projekt eines der bisher größten und wichtigsten dar. Da das Jobportal weiterhin durch das Medienkombinat gewartet wird, ist es notwendig, die Qualität über die Lebensdauer des Projektes hinweg zu gewährleisten. Dieser Prozess wird durch die Komplexität des Systems erschwert.

Die Entwicklung seitens des MedienKombinats umfasst u.a. die Programmierung passender TYPO3-Extensions nach dem MVC Muster. Die Programmierarbeiten werden dabei von allen Entwicklern lokal durchgeführt und in einem SVN Repositorium zusammengeführt. Welcher Code in das SVN Repositorium übertragen wird, obliegt der Erfahrung und Verantwortung der Entwickler. Darüber hinaus gibt es die Entwicklungsumgebung Alpha und Beta, welche Testdatenbanken mit Abzügen aus der Liveumgebung bereitstellen. Auf die Alpha Umgebung können lediglich Mitarbeiter des MedienKombinats zu greifen. Sie dient dazu, Integrationstests vorzunehmen und Änderungen, welche auf den weiteren Umgebungen durchgeführt werden, zu testen. Die Beta Umgebung ermöglicht es dem Kunden, Änderungen, welche auf das Livesystem übertragen werden sollen, zu prüfen und freizugeben. Kritische Änderungen seitens der Entwicklungsabteilung werden nur nach Absprache mit der Projektleitung veröffentlicht und teilweise automatisiert durch ein Ant Skript übertragen.

Abbildung 3–1: Hogapage.de

3.2 Anforderungsanalyse

Erfahrungen innerhalb des MedienKombinats haben gezeigt, dass manuelles Testen der Software und Gewährleistung der Qualität bereits in kleineren Projekten kosten- und zeitintensiv ist. Um zukünftig Fehler frühzeitig zu identifizieren und damit die Fehlerkorrekturphasen erheblich zu reduzieren, sollen die Möglichkeiten der Qualitätssicherung in Softwareprojekten evaluiert werden. Das Resultat soll qualitativ hochwertigere und schneller entwickelte Software sein, um Kosten zu sparen und sich besser am Markt platzieren zu können.

Dazu ist es erforderlich, den Stand der Technik im Bereich des Testens von Software zu analysieren und erste qualitätssichernde Maßnahmen am Beispiel des vorher beschriebenen Webportals zu etablieren.

Aus dieser Aufgabenstellung resultieren vielfältige Zielstellungen und Anforderungen für diese Diplomarbeit. Nach der Analyse des Standes der Technik werden Lösungsansätze aufgezeigt, um die gewonnenen Erkenntnisse in dem bereits erwähnten TYPO3 Projekt, dem Hoga Jobportal, zu erproben. Dabei gilt es zu beachten, dass eine vollständige Kompatibilität zu TYPO3 bzw. PHP und die Integration in die TYPO3-Umgebung erforderlich sind. Die erarbeiteten Lösungsansätze werden anschließend auf einer extra aufgesetzten

Testumgebung hinsichtlich ihrer Tauglichkeit für das Entwicklungsvorgehen und die Unternehmenskultur des MedienKombinats examiniert.

Da die Entwicklungsarbeiten gemäß dem Wasserfall-Modell bereits begonnen haben und die Softwarearchitektur entworfen ist, werden nur vereinzelte Maßnahmen, wie automatisierte Tests, fest in das Entwicklungsvorgehen integriert. Somit wird das Projekt nicht zu stark beeinflusst, um den Erfolg nicht auf Grund eines erhöhten Entwicklungsaufwands zu mindern. Die Maßnahmen müssen dementsprechend schnell und ohne große Umstellungen bzw. Aufwand integrierbar sein. Darüber hinaus müssen sie für alle Projektbeteiligten transparent und verständlich gestaltet sein.

Zu beachten ist, dass es nicht möglich ist, eine gesamte Qualitätssicherungsabteilung innerhalb des MedienKombinats aufzubauen. Dementsprechend müssen Maßnahmen gewählt werden, wie beispielsweise testgetriebene Entwicklung, um Programmierer eigenständig testen zu lassen.

Auf Basis der Ergebnisse dieser Implementierung erster qualitätssichernder Maßnahmen sollen weitere erforscht und in Form eines Konzepts konserviert werden. Damit steht dieses für zukünftige Projekte zur Verfügung, um langfristig qualitätssichernde Maßnahmen zuverlässig planen und schnellstmöglich integrieren zu können. Der Bedarf nach einem weiterführenden Konzept gründet auf dem Zeitrahmen der Diplomarbeit. Dieser ermöglicht es nicht, ein umfassendes und vollständiges Modell zur Qualitätssicherung zu etablieren. Das Modell soll durch dieses Konzept vorbereitet werden.

Mit den bisher genannten Anforderungen und Zielen sind eine Sensibilisierung und Schulung des Personals hinsichtlich der Thematik und evtl. kommender Umstellungen verbunden.

Wie in Kapitel 1 bereits erwähnt, setzt das MedienKombinat, mit Blick auf die Wahl der Werkzeuge, auf Open Source Software. Dementsprechend sollen eingesetzte Testwerkzeuge frei verfügbar sein um diese ggf. erweitern zu können. Lizenzpflichtige Varianten stehen außer Betracht, um Kosten zu sparen.

4 Lösungsansätze

Nachdem im vorherigen Kapitel betrachtet wurde welche Ziele diese Arbeit erreichen soll, werden im Folgenden Lösungsansätze diskutiert. Dies umfasst die Anforderungen zu evaluieren wie Tests erstellt und automatisiert ausgeführt werden können. Weiterhin steht im Mittelpunkt der Betrachtung, welche Vorarbeiten nötig sind und wie der Testprozess in das Entwicklungsvorgehen des Hoga Jobportals integriert werden kann. Abschließend werden verschiedene Testwerkzeuge hinsichtlich ihrer Features und Eignung für den Einsatz im Projekt betrachtet.

Auf Grund der Größe des Projekts, der Unternehmenskultur und des Entwicklungsvorgehens sind dabei erprobte und standardisierte Verfahren und Werkzeuge eigens entwickelten oder kostenpflichtigen vorzuziehen.

4.1 Test-Modell

Wie in Kapitel 2.2 betrachtet, gibt es verschiedene Softwareentwicklungsmodelle, welche die Testarbeiten in den Entwicklungsprozess einordnen und hilfreich sind, um den Testprozess zu modellieren. Welche für das Projekt des Hoga Jobportals in Frage kommen wird im Folgenden analysiert. Die eigentliche Planung der Testphasen wird in Kapitel 5.2 näher betrachtet. Für die Modellierung des Testprozesses sind in der Praxis bewährte und standardisierte Verfahren zu bevorzugen, damit weniger Probleme während der Einführungsphase auftreten und schablonenartige Problemlösungen für die Testarbeiten zur Verfügung stehen.

Aus diesem Grund ist das in Kapitel 2.2.1 skizzierte allgemeine V-Modell attraktiv. Bei diesem wird zeitnah, zu jeder Entwicklungsphase korrespondierend, eine Testphase durchgeführt, um die Entwicklungsergebnisse validieren und verifizieren zu können. Entwickler, die an der direkten Programmierung eines Projektes beteiligt sind, übernehmen dabei lediglich eine beratende Funktion hinsichtlich der Erstellung von Tests. Daraus resultiert ein erhöhter Personalaufwand für Tester, aber Fehler können mitunter schneller behoben werden auf Grund der Nähe zur entsprechenden Entwicklungsphase. Dies ist ein wichtiger Vorteil gegenüber einem Testprozess, der vollkommen entkoppelt vom Entwicklungsprozess durchgeführt wird, wie es bei einem Entwicklungsvorgehen nach dem klassischen Wasserfall-Modell der Fall ist.

Eine Alternative besteht in der agilen Softwareentwicklungsmethode XP. (vgl. Kapitel 2.2.2) Bei dieser werden ganz im Gegensatz zum V-Modell Iterationszyklen basierend auf User-Storys durchgeführt. Der Programmcode wird dabei von den Entwicklern selbst durch Unit-Tests abgedeckt. Darüber hinaus erstellen sie Akzeptanztests, um die Oberfläche zu testen. Allerdings werden Tests wie Performanz- und Lasttest nicht berücksichtigt. Hinzu kommt, dass der Kunde erheblich mehr in den Entwicklungsprozess involviert wird. Dementsprechend müsste das gesamte Entwicklungsvorgehen innerhalb des Projektes umgestellt werden, weshalb ein Softwareentwicklungsvorgehen nach XP nicht in Frage kommt.

Für das MedienKombinat bietet sich eine Mischform aus dem V-Modell und XP an, um den Testprozess zu modellieren. Da das Entwicklungsvorgehen innerhalb des Hoga Jobportals dem Wasserfall-Modell entspricht, kann dieses leicht um einen Testprozess gemäß dem V-Modell erweitert werden. Das Wasserfall-Modell besteht aus dem linken Ast des V-Modells (vgl. Abbildung 2–2), womit Umstellungen innerhalb des Entwicklungsvorgehens während der Etablierung des Testprozesses gering ausfallen. Allerdings sollen keine weiteren Tester zum Projekt hinzugezogen werden. Daher ist der Aspekt der testgetriebenen Entwicklung aus XP interessant, um Tests in der Komponententestphase zu generieren. Somit ist jedoch eine Sensibilisierung und Schulung der Entwickler hinsichtlich testgetriebener Entwicklung von Nöten. Die Planung der einzelnen Testphasen wird in Kapitel 5.2 behandelt. Im Folgenden werden die Grundzüge testgetriebener Entwicklung diskutiert.

4.2 Testgetriebene Entwicklung

„The secret in testing is in writing testable code“

- Misko Hevery

Ein essentieller Aspekt beim Testen von Software ist so früh wie möglich und automatisiert, d.h. wiederholbar, zu testen. Nach dem allgemeinem V-Modell (vgl. Kapitel 2.2.1) stellt die Komponententestphase den Startpunkt der Testarbeiten dar. Diese Teststufe findet zeitgleich mit der Entwicklungsphase statt. Ein noch relativ neuer Ansatz, um Tests in dieser Phase zu entwerfen, ist die testgetriebene Entwicklung, welche eine Software-Design-Methode ist. Diese kommt u.a. bei der agilen Software-Entwicklung zum Einsatz und

verlangt, dass Unit-Tests (Komponententests) nicht von Testern, sondern von den Entwicklern selbst korrespondierend zum Entwicklungsprozess geschrieben werden. Das hat den Vorteil, dass keine Testabteilung innerhalb des Unternehmens benötigt wird. Testgetriebene Entwicklung dient aber nicht nur dazu, die Software zu testen, sondern um das Design und somit auch das Verhalten besser zu spezifizieren. Außerdem werden in späteren Teststufen Akzeptanztests geschrieben für den Integrations- und Systemtest.

Laut (vgl. [Wes06], S. 2f) sind bei der testgetriebenen Entwicklung drei Direktiven zu beachten. Diese sind:

- Motiviere jede Änderung des Programmverhaltens durch einen Unit-Test (vgl. Kapitel 5.3.1)
- Bringe den Code immer in die einfachste Form
- Integriere den Code so häufig wie möglich

Testarbeiten in der Integrationsstufe haben zum Teil das Problem, dass der Quellcode nur sehr schwer oder nahezu untestbar ist. Die Idee der testgetriebenen Entwicklung ist, Tests zu schreiben, bevor der zugehörige Code entwickelt wird und jede Änderung am Code durch Tests zu motivieren. Dadurch kann das oben beschriebene Problem nicht auftreten, da das Code-Design maßgeblich beeinflusst wird. Die Tatsache, dass bereits Tests für den Code geschrieben wurden, beweist, dass dieser testbar ist. Mit welchem Werkzeug diese Tests erstellt und ausgeführt werden, wird im übernächsten Abschnitt und wie dieser Prozess in der Praxis aussieht in Kapitel 5.3.1 betrachtet.

Die Anforderung für einen Test wird z.B. entweder in User-Stories abgebildet (vgl. Kapitel 2.2.2), einem Fehlerreport, dem Testplan oder dem Lastenheft entnommen. Dabei sollte es pro Anforderung oder Fehler einen Test geben. Die Anforderungen sind die gleichen, die beim Schreiben des eigentlichen Programmcodes herangezogen werden würden ohne Tests. Nach (vgl. [Wes06], S. 2) müssen die Tests zu Beginn natürlich fehlschlagen, da der zu testende Programmcode noch gar nicht existiert. Wird nun anschließend der zugehörige Programmcode erstellt, damit der Test nicht mehr fehlschlägt, wird auch nur der Programmcode erstellt, welcher notwendig ist, um die jeweilige Anforderung zu erfüllen. Abschließend wird ein Refactoring durchgeführt, um eventuelle Code-Duplikationen und ähnliches zu beseitigen. Dadurch entsteht ein sauberes und evolutionäres Design, das inkrementell voran getrieben wird. Zuerst wird es spezifiziert, danach programmiert und

abschließend verifiziert. Zudem kann es schneller entstehen, da sich ein Entwickler nur auf die aktuelle Anforderung konzentriert.

Weil sich Software permanent in einem Änderungsprozess befindet und zudem unter Entropie leidet, sind ständige Refactorings im Software-Lebenszyklus unerlässlich, um die strukturelle Software-Qualität aufrecht zu erhalten und den Programmcode ständig in die einfachste Form zu bringen. Um diese Refactorings mit Sicherheit und Vertrauen durchführen zu können, wird ein Sicherheitsnetz benötigt, da es insbesondere bei komplexen Software-Projekten nicht abzuschätzen ist, welche Auswirkung eine Änderung auf andere Programmteile haben kann. Sind alle Programmteile durch Tests abgedeckt und die Tests decken wiederum alle Anforderungen ab, ist es lediglich notwendig, nach einer Änderung alle Tests auszuführen, um eine evtl. eingeschlichene Fehlerwirkung aufzudecken. Dadurch ist der Eintritt einer unabsehbaren Auswirkung durch eine Änderung relativ unwahrscheinlich, da die Tests beweisen, dass die Software wie bisher funktioniert. Die Voraussetzung ist selbstverständlich immer, dass die Entwickler sorgfältig entwickeln und ihren Programmcode durch Tests abdecken.

Das Erstellen der Unit-Tests auf diese Weise ist weder den Whitebox-(vgl. Kapitel 2.3.2) noch den Blackbox-Verfahren (vgl. Kapitel 2.3.1) zuzuordnen sondern bildet eine Misch-Form, die den Greybox-Verfahren angehört. Zu Beginn ist es ein Blackbox-Verfahren, da der Programmcode noch gar nicht existiert und Testfälle lediglich auf Anforderungen basieren. Im Zuge des Refactorings wird es ein Whitebox-Verfahren, da der Programmcode beim Erstellen oder Ändern der Tests heran gezogen wird.

Nach (vgl. [Wes06], S. 3ff) ist, speziell bei Entwicklungsarbeiten im Team, eine kontinuierliche Integration notwendig, bei der das Testen in den Entwicklungsprozess integriert ist. So müssen z.B. nach jeder Änderung im SVN alle Tests ausgeführt werden, um eventuelle Konflikte von Programmteilen untereinander frühestmöglich auszudecken und die funktionale Qualität der Software aufrecht zu erhalten.

Ein weiterer Vorteil von testgetriebener Entwicklung ist der greifbare Fortschritt. Besonders wenn die Software noch nicht fertig ist und vom Kunden oder Projektmanager nicht gesichtet werden kann, bieten Tests eine gute Möglichkeit, den aktuellen Projektstand abzubilden. Das Ergebnis aller Tests gibt Auskunft darüber, welche Anforderungen bereits realisiert wurden und welche Programmteile fehlerfrei funktionieren. Eine Möglichkeit, die

Tests automatisiert nach jeder Code-Integration auszuführen und die Ergebnisse zu kommunizieren wird in Abschnitt 4.5 in Form von Continous Integration betrachtet.

Zusammenfassend lässt sich das Vorgehen bei testgetriebener Entwicklung in drei Schritte gliedern:

- Einen Test für eine Anforderung, wie Fehlerkorrektur oder Implementierung einer neuen Funktionalität, schreiben, welcher vorerst nicht erfüllt wird.
- Die Problemstellung in Form des einfachsten Programmcodes lösen, mit möglichst wenig Aufwand.
- Abschließendes Refactoring - das umfasst u.a. den Code mehr zu abstrahieren oder Wiederholungen zu entfernen, um ihn verständlicher zu gestalten.

Mit welchem Werkzeug Unit-Tests in der testgetriebenen Entwicklung im Bereich der Komponententeststufe erstellt und ausgeführt werden, wird im übernächsten Abschnitt aufgezeigt.

4.3 Nachteile von testgetriebener Entwicklung

Eine der größten Nachteile testgetriebener Entwicklung ist, dass Entwickler ihre Fehler leicht übersehen können. Aus diesem Grund hat sich z.B. bei XP der Ansatz der paarweisen Programmierung etabliert, um diesem Sachverhalt entgegenzuwirken.

Weiterhin setzt der Einsatz testgetriebener Entwicklung Erfahrungen und Know-how in diesem Bereich voraus, was einen erhöhten Aufwand in der Einführungsphase nach sich zieht. Ohne Erfahrung kann es schwer zu verstehen sein, wie etwas getestet werden soll, das noch gar nicht programmiert ist. Dies kann dazu führen, dass die Prinzipien der testgetriebenen Entwicklung nicht vollends befolgt werden, was wiederum den gesamten Entwicklungsprozess gefährdet. Werden nicht genügend Tests geschrieben, um die geforderte Testabdeckung zu erreichen, sind Refactorings schwieriger möglich, wodurch die Software-Qualität und das Design stark leiden.

Aber selbst wenn der geforderte Abdeckungsgrad erreicht ist, ist noch keine Fehlerfreiheit garantiert. Externe Fehlerquellen wie ein Serverausfall oder nicht ansprechbare Schnittstellen aber auch die fehlende Möglichkeit, einen Test mit allen möglichen Testdaten auszuführen, werden nicht berücksichtigt.

Allgemein wird oftmals falsch verstanden, dass nicht das Testen von Klassen im Mittelpunkt steht, sondern das Design der Software zu spezifizieren. Dieses Missverständnis bedingt ein Übermaß an sinnlosen Tests, welche Klassen, aber nicht das Verhalten der Software testen.

4.4 Die xUnit Familie

Um Unit-Tests bequem erstellen und ausführen zu können, gibt es für eine Vielzahl von Programmiersprachen xUnit-Frameworks, deren Verwendung eine immer größere Verbreitung aufweist. Dies liegt an den zahlreichen Funktionen, welche sie bieten, um Programmcode einfach und automatisch testen zu können. Dabei wird ein Test geschrieben, der das zu testende Modul z.B. auf der Kommandozeile aufruft und dessen Verhalten oder Antwort evaluiert. Vertreter sind u.a. JUnit für Java, NUnit für .NET oder PHPUnit für PHP.

Das xUnit Framework, welches als Beispiel dient, ist PHPUnit von Sebastian Bergmann, da TYPO3, und somit auch das Projekt dieser Arbeit, PHP-basiert ist. In Abschnitt 4.6.2 wird dieses einem weiteren Vertreter der xUnit-Frameworks aus dem PHP-Umfeld gegenüber gestellt.

Im Folgenden werden die allgemeinen Funktionalitäten eines xUnit-Frameworks anhand von PHPUnit kurz analysiert. In Abschnitt 5.3.1 werden einige anhand eines Beispiels aus der Praxis demonstriert. Eine umfangreiche Einführung in PHPUnit mit Beispielen für alle Funktionalitäten ist auf (vgl. [URL:PHPUnit]) zu finden.

Eines der häufigsten Probleme, welches auftritt während Tests geschrieben werden, ist die Testumgebung vor und nach dem Test in einen bestimmten Zustand zu versetzen. PHPUnit bietet herfür die Möglichkeit der Fixtures. Dies wird realisiert, indem es die Methoden setUp und tearDown in der Basis-Testfall-Klasse gibt, aus der die eigene Testfall-Klasse abgeleitet wird, womit die beiden Methoden mit eigenen Funktionalitäten überschrieben werden können. SetUp wird vor jedem Test ausgeführt, wodurch z.B. Datenbank-Verbindungen aufgebaut werden können und tearDown nach jedem Test, wodurch die Datenbank z.B. wieder geleert wird.

Es gibt noch eine Vielzahl von Features, die im Folgenden nur kurz genannt und erläutert werden:

- Assertions: Durch verschiedene assert-Methoden ist es einfach zu prüfen, ob ein Wert true ist, zwei Werte gleich sind und vieles mehr.

- Exceptions: Es kann getestet werden ob Exceptions geworfen wurden.
- Data Providers: Dem Testfall kann eine Funktion bekannt gegeben werden, die ein Array zurückgibt. Für jeden Wert dieses Arrays wird der Testfall ausgeführt. Dies bietet sich für einen Testfall, der nach der Grenzwertanalyse (vgl. Kapitel 2.3.1.2) entworfen wurde, an
- Stubs und Mocks: Da die Unit-Tests isoliert von an anderen Programmteilen sein sollten, zumindest wenn die Programmteile nicht von Tests abgedeckt sind, aber die zu testende Funktionalität von einem anderen Teil abhängen kann, bietet PHPUnit die Möglichkeit, diese Programmteile zu simulieren. Dazu wird über die getMock Methode ein Dummy einer beliebigen Klasse erstellt. Dieser kann Methoden und Klassen-Variablen besitzen, deren Wert und Verhalten definiert wird. So kann z.B. überwacht werden, wie oft und mit welchen Parametern Methoden aufgerufen wurde. Durch Stubs und Mocks können auch Funktionalitäten getestet werden, die von noch nicht existenten Programmteilen abhängen.
- Datenbanken: PHPUnit bietet eine spezielle Basis Datenbank-Testfall-Klasse, aus welcher eigene Testfälle abgeleitet werden können. Diese Klasse bietet Methoden, um vor einem Test eine Datenbankverbindung aufzubauen, danach Testdaten einzufügen, den erwarteten und tatsächlichen Inhalt der Datenbank in Testfällen zu prüfen und abschließende Aufräumarbeiten auszuführen. Die Daten können in verschiedenen Formaten wie CSV oder XML vorliegen, wodurch die Implementierung sehr bequem ist und Datenbanken schnell aufgesetzt und in einen bekannten Zustand versetzt werden.
- Report: Nachdem ein Test ausgeführt ist, ist es erstrebenswert, die Ergebnisse zu loggen, zu publizieren und zu archivieren. PHPUnit bietet hierfür die Möglichkeit, die Ergebnisse in verschiedene Formate wie dem Testdox-Format oder einem JUnit-XML-Report zu exportieren. Dadurch kann z.B. der Projektmanager oder Kunde sehen wie sich die Testarbeiten über den Projektverlauf entwickelt haben oder wie viele Tests aktuell fehlschlagen.
- Metriken: Um zu überwachen wie viel Programmcode von Tests abgedeckt ist, d.h. welche Güte die Tests aufweisen, bietet PHPUnit die Möglichkeit, während der Testausführung zu analysieren welche Programmteile erreicht wurden und die Ergebnisse in Form eines Code Coverage Reports zu exportieren. Dieser bedient dabei eine Vielzahl von Code-Überdeckungsverfahren und weitere Statistiken. Dazu zählen u.a. die in Kapitel 2.3.2 betrachtete Anweisungs-, Zweig- und Bedingungsüberdeckung aber auch die Anzahl der

Klassen oder die Komplexitätskenngröße nach McCabe. (vgl. [URL:Metrics]) Das Beispiel eines Code Coverage Reports ist in Kapitel 5.3.1 zu finden.

PHPUnit besitzt noch eine große Anzahl weiterer Features, die aber hier nicht weiter Beachtung finden. Die Vielzahl der Features bedingt auch die Entscheidung zum Einsatz eines etablierten xUnit Frameworks anstatt ein eigenes zu entwickeln.

4.5 Continous Integration

Laut (vgl. [Duv07], S. 23ff) hat Software Entwicklung oftmals das Problem der Annahmen. So wird z.B. angenommen, dass Code wie gewünscht funktioniert oder Coding Standards eingehalten werden. Aber ohne dies permanent und kontinuierlich zu prüfen, steigt das Projektrisiko und die Software Qualität leidet. Eine Möglichkeit, diese Prüfung automatisch und wiederholt vorzunehmen, stellt ein Continous Integration Server oder kurz CI-Server dar. Dieser bildet eine Schnittstelle zwischen dem entwickelten Code jedes einzelnen Entwicklers und der auslieferbaren Software. In Verbindung mit verschiedenen Continous Integration Praktiken kann so den eingangs genannten Problemen entgegengewirkt werden.

Die Prüfung kann z.B. jedes Mal stattfinden wenn Änderungen im SVN Repositorium vorgenommen wurden, womit sich ein CI-Server hervorragend eignet um änderungsbezogene Tests und Regressionstests durchzuführen. (vgl. Kapitel 2.2.3.3) Dadurch können der Verlauf, der aktuelle Stand und die Qualität des Projekts in Form von Builds überwacht werden. Ein Build ist die eigentliche Prüfung und umfasst verschiedene Aktivitäten, um die Software zu testen, zu inspizieren und zu deployen, d.h. zur Nutzung bereitzustellen. Typischerweise werden dafür Werkzeuge wie Ant für Java, Rake für Ruby oder Phing für PHP verwendet, um die Build-Skripte auszuführen. Continous Integration besteht somit aus den Teilen kontinuierliches Testen, kontinuierliche Inspektionen, kontinuierliches Deployment und kontinuierliches Feedback.

Der Continous Integration Zyklus lässt sich dabei wie folgt skizzieren. (vgl. Abbildung 4–1) Ein Entwickler überträgt neuen Code in das SVN Repositorium worauf hin der CI-Server diese Änderungen kurze Zeit später bemerkt. Es wird ein Build gestartet, wobei der neueste Stand aus dem SVN Repositorium geholt wird und verschiedene Test- sowie Inspektionstools ausgeführt werden. Das Ergebnis wird anschließend in Form eines Berichts publiziert und ist z.B. durch ein Web-Interface einsehbar. Welche Tools das umfasst und welche weiteren Möglichkeiten ein Build bietet wird in Abschnitt 4.6.1 betrachtet, während der Analyse verschiedener Vertreter von CI-Servern. Die Integration eines CI-Servers in den Entwicklungsprozess des Hoga Jobportals wird in Kapitel 5.2.1 behandelt.

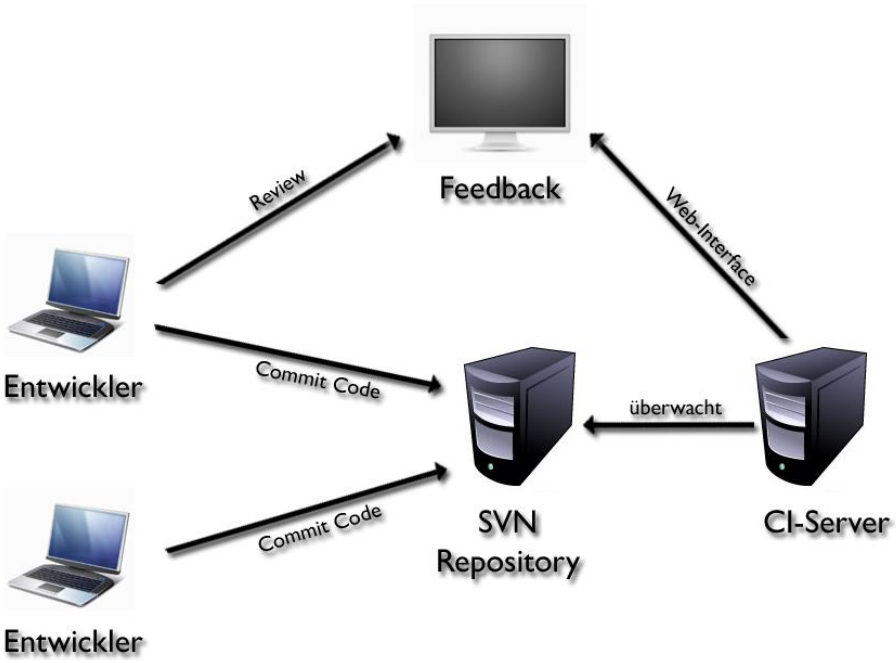


Abbildung 4–1: Continous Integration Zyklus (vgl. [Duv07], S.26)

Um einen CI-Server sinnvoll nutzen zu können, müssen nach (vgl. [Duv07], S. 39ff) verschiedene Continous Integration Praktiken beachtet werden. Eine essentielle Bedingung ist u.a., dass jeder Entwickler regelmäßig seinen Code in das SVN Repositorium überträgt. Regelmäßig heißt dabei mehrmals pro Tag und der Code muss von genügend vielen Tests abgedeckt sein. Bestehen nicht genügend Tests sind die Testergebnisse wenig aussagekräftig, womit der Nutzten stark eingeschränkt wird. Aus diesem Grund ist es ratsam ein Tool zu integrieren, welches einen Code Coverage Report erstellt. Somit kann die Testabdeckung überwacht werden. Wenn ein Build fehlschlägt besteht die oberste Priorität darin, den Fehler zu beheben damit es wieder erfolgreich ist, was bedeutet, dass alle Tests

und Inspektionen bestanden wurden. Sind Builds immer erfolgreich, besteht nicht die Gefahr, dass Entwickler mit Code arbeiten, der möglicherweise fehlerhaft ist.

Wenn diese Bedingungen erfüllt sind, bietet Continous Integration viele Vorteile. So wird das Projektrisiko minimiert, indem die Software bei jeder Änderung automatisch getestet wird und Fehler dadurch frühestmöglich aufgedeckt werden. Dabei wird die in Abschnitt 4.2 betrachtete testgetriebene Entwicklung integriert, unterstützt und gefördert, da Unit-Tests ein vitaler Aspekt für den Nutzen eines CI-Servers sind. Szenarien wie: „Bei mir lokal funktioniert alles“, haben keine Berechtigung mehr, da die Software durch den CI-Server permanent integriert wird und Feedback über das Resultat vorhanden ist. Aufgaben wie das Testen oder Deployment der Software, die sich während des Projektverlaufs stetig wiederholen werden automatisiert ausgeführt. Die Ausführung kann dabei von jedem Ort und zu jedem Zeitpunkt erfolgen z.B. via Web-Interface. Nicht zu unterschätzen ist auch das gesteigerte Vertrauen in die entwickelte Software, da deren Stand und Qualität permanent einsehbar sind und verfolgt werden können. Die Aussagen über den Stand und die Qualität basieren dabei nicht auf Annahmen sondern auf Fakten.

4.6 Testwerkzeuge

Um die bisher diskutierten Lösungsansätze umzusetzen, bedarf es softwareseitiger Unterstützung. Es ist als erstes ein Continous Integration Server zu wählen. Dieser dient als zentrale Schnittstelle, um weitere Werkzeuge in den Entwicklungsprozess integrieren zu können und die Ergebnisse dieser zu publizieren. Darüber hinaus wird ein Testframework benötigt, um Unit- und Akzeptanztests auszuführen. Für das Erstellen der Akzeptanztests ist ein eigenständiges Werkzeug notwendig. Das gleiche gilt für Performanztests. Im Folgenden werden verschiedene Vertreter hinsichtlich ihrer Eignung für das Projekt des Hoga Jobportals betrachtet. Gemäß der Philosophie des Medienkombinats wird dabei darauf geachtet, dass es sich um Open Source Software handelt.

4.6.1 Continous Integration Server

Der Nutzen und prinzipielle Einsatz eines Continous Integration Servers wurde in Abschnitt 4.5 erläutert. Es gibt zahlreiche Vertreter und im Folgenden soll die Wahl für einen getroffen werden, um ihn für das Hoga Jobportal zu integrieren.

Der Deploymentprozess wird bisher manuell durch ein Ant-Skript, sprich durch ein Build Tool, ausgeführt. Genau solche Build Tools werden auch eingesetzt, um den Buildprozess eines Continous Integration Server zu konfigurieren. Daher ist es wünschenswert, dass der Continous Integration Server das bereits bestehende Ant-Skript nutzen kann.

Nach langwieriger Evaluierung sind drei Vertreter in die engere Auswahl gerückt. Dazu zählen Hudson, CruiseControl in Verbindung mit phpUnderControl und Xinc. Sowohl Hudson als auch CruiseControl sind Java-basiert im Gegensatz zu Xinc, welches durchgehend in PHP programmiert wurde. Somit ist Xinc eigentlich der geeignetste Vertreter, allerdings ist er auch der jüngste und somit nicht so ausgereift wie Hudson oder CruiseControl. Als Build Tool kommt Phing zum Einsatz aber es gibt zum Beispiel keine Möglichkeit, ein Build manuell in der Oberfläche anzustoßen. Auch ist die Dokumentation unzureichend und die Community noch zu klein, um beispielweise Erweiterungen zu erstellen oder Hilfe zu bieten. Deshalb ist Xinc nicht optimal für den Einsatz im Projekt des Hoga Jobportals. Sowohl für Hudson als auch für CruiseControl gibt es eine ausführliche Dokumentation und ein große Community. Builds können mit Ant-, Maven- oder Phing-Skripten und noch weiteren konfiguriert werden. Somit kann das bestehende Deployment-Skript bei Hudson und bei CruiseControl eingebunden werden. Beide bieten darüber hinaus eine große Anzahl an Features, wie das Starten eines Builds in Abhängigkeit von z.B. einer Änderung in einem SVN Repositorium oder der Versand von Emails, abhängig vom Ergebnis eines Builds. Weiterhin besteht eine gute Unterstützung, um Werkzeuge aus dem PHP Umfeld zu integrieren. Dazu zählen Werkzeuge wie PHPDoc, PHP Code Sniffer, PHP Copy & Paste Detector, PHP Mess Detector um die Qualität der Software zu prüfen und zu dokumentieren aber auch ein xUnit Framework. Der Unterschied ist, dass diese bei Hudson in Form von in Java geschriebenen Plugins und bei CruiseControl durch phpUnderControl eingebunden werden, welches ein PHP Plugin ist. Durch phpUnderControl ist es einfacher ggf. Anpassungen vorzunehmen. Da TYPO3 und somit das gesamte Projekt PHP-basiert ist, stellt CruiseControl in Verbindung mit phpUnderControl den interessantesten Vertreter dar und soll genutzt werden.

Kriterium	Xinc	Hudson	CruiseControl + phpUnderControl
Programmiersprache	PHP	JAVA	JAVA + PHP
Nativ Unterstützte Build-Script Werkzeuge	Phing; um weitere erweiterbar	Maven, Ant, Shell Script, Windows Batch Command; um weitere erweiterbar	Ant, NAnt, Maven, Phing, Rake, Xcode; um weitere erweiterbar
Dokumentation	Unzureichend; zum Zeitpunkt dieser Diplomarbeit nicht erreichbar	Ausführlich	Ausführlich
Community	Klein	Groß	Groß
Oberfläche	Gut strukturiert; wenig Informationen abrufbar	Gut strukturiert; sehr viele Informationen abrufbar; Möglichkeit neue Projekte einzurichten und zu bearbeiten	Sehr gut strukturiert; phpUnderControl Oberfläche speziell an Bedürfnisse von PHP angepasst; sehr viele Informationen abrufbar
Unterstützung für diverse PHP Werkzeuge	Native Unterstützung für PHPUnit, PHPMD, PHPDoc, PHP CodeSniffer und weitere	Plugins für PHPUnit, PHPMD, PHPDoc, PHP CodeSniffer und weitere	phpUnderControl für PHPUnit, PHPMD, PHPDoc, PHP CodeSniffer und weitere
Weiterentwicklungsstatus	Nicht aktiv weiterentwickelt	Aktiv weiterentwickelt	Aktiv weiterentwickelt
Automatische Builds	Ja	Ja	Ja
Manuelle Builds	Nein	Ja	Ja

Tabelle 4-1: Bewertung der verschiedenen Vertreter von Continous Integration Servern

Dabei dient CruiseControl als der eigentliche Continous Integration Server, um ein Build auszuführen und die Ergebnisse zu publizieren. Das von Manuell Pichler entwickelte phpUnderControl bereitet die Ergebnisse der o.g. PHP Werkzeuge auf und bietet eine gut strukturierte Oberfläche, um diese darzustellen. (vgl. Abbildung 4–2)

Abbildung 4–2: phpUnderControl Oberfläche

Die genaue Konfiguration und Integration in den Entwicklungsprozess wird in Kapitel 5.2.1 betrachtet.

4.6.2 xUnit Framework

Der Grund und Nutzen eines xUnit Frameworks wurde bereits in Kapitel 4.2 und 4.4 erläutert. Da das Hoga Jobportal durchweg in PHP programmiert wird, ist ein PHP-basiertes xUnit Framework von Nöten. Es gibt im PHP Umfeld zwei nennenswerte Vertreter. Zum einen Simple Test und zum anderen das bereits erwähnte PHPUnit. Sie bieten die im Abschnitt 4.4 beschriebenen Features und unterscheiden sich darin nur marginal. Simple Test bietet im Gegensatz zu PHPUnit eine Weboberfläche, um Tests auszuführen. Allerdings wird es nur sehr langsam weiterentwickelt und die Community ist noch klein. Dagegen findet PHPUnit eine sehr große Verbreitung und wird durch Werkzeuge wie CruiseControl oder Xinc gut integriert. Interessant ist auch die Möglichkeit, dass Selenium Tests (vgl. Abschnitt 4.6.3) integriert und ausgeführt werden können. Darüber hinaus wird es aktiv weiterentwickelt und ist sehr gut dokumentiert. Somit ist PHPUnit der de facto Standard im PHP Umfeld und wird als xUnit Framework eingesetzt. Wie es genau genutzt wird und in TYPO3 integriert ist, wird in Kapitel 5.3.1 betrachtet.

4.6.3 Capture und Replay Werkzeuge

Um Oberflächentests auf der Akzeptanzteststufe automatisiert durchführen zu können eignen sich xUnit Frameworks nur sehr bedingt. Es werden s.g. Testroboter oder auch Capture and Replay Werkzeuge benötigt. Diese ermöglichen es dem Tester z.B. die Interaktion auf einer Webseite aufzuzeichnen. Das umfasst, welche Seiten aufgerufen,

welche Elemente angeklickt oder auch welche Formulare abgeschickt wurden. Währenddessen können Prüfungen integriert werden, ob beispielsweise der korrekte Seitentitel auftaucht oder ein Formular korrekt validiert wurde. Auf diese Weise ist es möglich, das genaue Nutzungsverhalten eines Webauftritts in einer Testsuite abzubilden, um dieses jederzeit reproduzieren und testen zu können. Ein großer Vorteil hierbei ist, dass keine Programmierkenntnisse vorausgesetzt werden da das Erstellen und Ausführen der Tests zum größten Teil von den Werkzeugen übernommen wird und somit simpel gestaltet ist.

Nachdem viele Vertreter dieser Gattung von Werkzeugen hinsichtlich ihrer Eignung für das Projekt des Hoga Jobportals geprüft wurden, scheinen zwei Vertreter am geeignetsten. Zum einen Canoo Web Test und zum anderen Selenium. Canoo Web Test und auch die Oberflächentests selbst sind Java-basiert und werden in einem eigenen Browser ausgeführt. Dadurch kann beispielweise auch der Inhalt eines PDF-Dokuments geprüft werden und Tests laufen stabiler und schneller. Allerdings macht dieser Umstand das Testen in Browsern wie Firefox oder Internet Explorer schwer. Im Gegensatz dazu werden Selenium Tests direkt im herkömmlichen Browser ausgeführt und können keine PDF-Dokumente oder Flash Webseiten testen. Das Testen in normalen Browsern ist durch Java Script Injektionen möglich. Die Tests selbst werden als Html Dateien angelegt, können aber auch als xUnit Testcase für Sprachen wie PHP exportiert werden. In Zusammenspiel mit der Multibrowser Unterstützung ist es somit möglich, eine mächtige Cross Browser Testsuite, beispielweise automatisiert über einen Continous Integration Server, ausführen zu lassen. Ebenfalls können die exportierten Testfälle erweitert werden, um in der Integrationsteststufe zum Einsatz zu kommen.

Da die Entwicklungsumgebung PHP- und MySQL-basiert ist, Cross Browser Testing eine hohe Priorität genießt und eine Integrationsmöglichkeit in PHPUnit besteht, soll Selenium zum Einsatz kommen. Im Folgenden werden kurz die Möglichkeiten und Funktionen von Selenium erläutert.

Selenium selbst ist eine Software Suite und die für diese Arbeit relevanten Bestandteile sind Selenium IDE, Remote Control bzw. Grid und Hub. Die Selenium IDE ist ein Firefox Addon welches dazu dient, Tests aufzuzeichnen, zu exportieren und durch den Selenium Server auszuführen zu lassen. Es besteht allerdings keine Möglichkeit, dass Tests wiederholt automatisch ausgeführt werden können. Dies kann erreicht werden, indem die Tests als

PHPUnit Selenium-Testcase exportiert werden. Dadurch ist es möglich, Tests automatisch als Unit-Test durch einen Continous Integration Server ausführen zu lassen. Damit ein Browser bereitgestellt wird, ist die Selenium Remote Control notwendig. Diese agiert als ein Selenium Server und kann auf einem beliebigen Rechner laufen. Um beliebig viele Remote Controls fernsteuern zu können, wird das Selenium Grid und Hub benötigt.

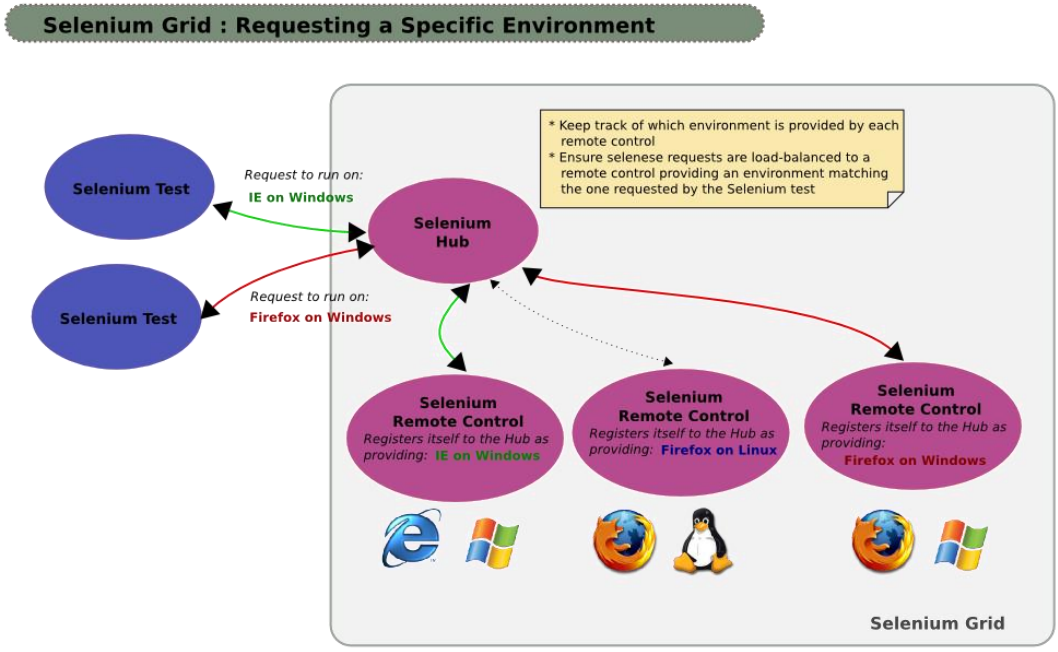


Abbildung 4–3: Selenium Grid (vgl. [URL:Grid])

Das Grid ist eine Erweiterung, um eine Vielzahl von Selenium Remote Controls betreiben zu können, was eine parallele und skalierbare Testausführung ermöglicht. Das Selenium Hub verwaltet die Selenium Server des Grids und dient als Schnittstelle zwischen den Server Instanzen und dem Test. Der Test bzw. das aufrufende xUnit Framework kann dabei beispielweise verschiedene Browser anfordern, in welchen der Test ausgeführt wird. Die genaue Funktionsweise aller Komponenten kann (vgl. [URL:Selenium]) entnommen werden.

Ein großer Nachteil von allen Capture and Replay Werkzeugen ist, dass sie keinen prüfenden Blick ersetzten. Es ist nicht möglich, das gesamte Layout einer Webseite zuverlässig zu testen. Auch sind die Tests im Gegensatz zu Unit-Tests sehr langsam. Bei entsprechender Größe der Testsuite und dem Ausführen in verschiedenen Browsern, kann ein Testdurchlauf schnell mehrere Stunden benötigen. Somit sind sie ungeeignet für das Ausführen innerhalb eines Continous Integration Zyklus‘ auf Basis von Änderungen innerhalb eines SVN

Repositoriums. Werden die Tests zu einer festen Zeit, beispielweise in der Nacht, ausgeführt, fällt dieses Problem weniger ins Gewicht.

4.6.4 Last Werkzeuge

Nachdem in den vorangegangen Abschnitten Werkzeuge für die Komponenten- und Akzeptanz- bzw. Integrationsteststufe betrachtet wurden, sind weiterhin Werkzeuge für die Performanz- und Lastteststufe notwendig. Es ist nicht möglich, hunderte oder tausende Nutzer eine Webapplikation testen zu lassen. Dafür sind Werkzeuge notwendig, welche die Last simulieren und die Stabilität bzw. das Antwortverhalten messen. Dabei gilt es, dazwischen zu unterscheiden, ob eine gesamte Webapplikation oder lediglich ein Webserver getestet wird. Um die Performanz eines Webservers zu ermitteln, ist ein Werkzeug wie httperf prädestiniert. Dieses lässt sich durch wenige Parameter auf der Kommandozeile konfigurieren und ausführen. Dabei werden simple Http-Anfragen an den zu testenden Server geschickt und dessen Antwortverhalten wird gemessen. Die Last ist mit Hinblick auf die Frequenz und gleichzeitige Anzahl der Anfragen skalierbar.

Um eine komplette Webapplikation hinsichtlich ihrer Performanz und Stabilität zu messen, ist ein Werkzeug wie httperf (vgl. [URL:Httperf]) insuffizient, da es keine virtuellen Nutzer simuliert. Es ist aber notwendig, geschäftsprozessbasierte Tests (vgl. Kapitel 2.2.3.1) durchzuführen, um die tatsächliche Interaktion reproduzieren zu können. Geeignet dafür ist beispielweise JMeter. (vgl. [URL:JMeter]) Damit können Tests ähnlich wie mit der Selenium IDE (vgl. Abschnitt 4.6.3) aufgezeichnet werden. Somit werden einzelne Nutzungsszenarien innerhalb der Webapplikation abgebildet. Innerhalb der Testausführung ist es nun möglich, die Anzahl und Frequenz der gleichzeitigen, virtuellen Nutzer vorzugeben und die Ergebnisse hinsichtlich einer Vielzahl von Eigenschaften zu messen. (vgl. Abbildung 4–4) Dazu zählt u.a. das Zeitverhalten aber auch die Rate an fehlerhaft ausgelieferten Antworten.

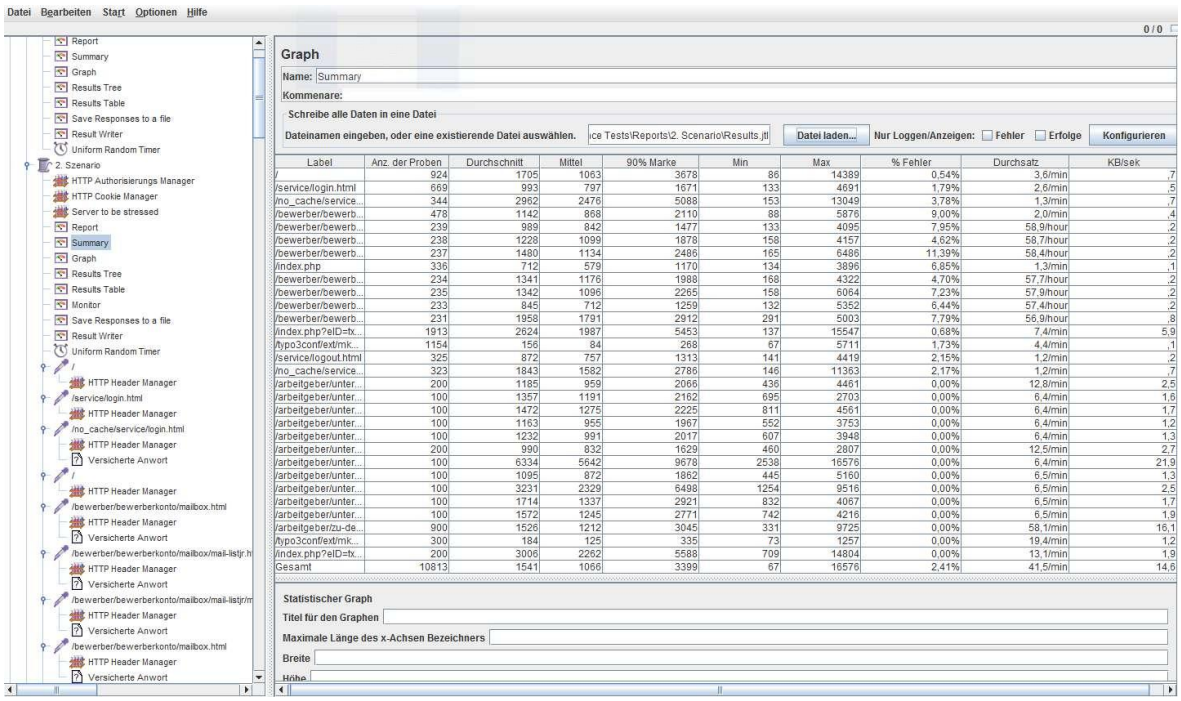


Abbildung 4–4: Übersicht JMeter

Ein Vorteil von sowohl httperf als auch JMeter ist der geringe Bedarf an Betriebsmitteln. Es ist möglich, eine realistische Last zu simulieren ohne zusätzliche Rechner zu benötigen.

5 Implementierung und weitergehende Konzeptionierung

Nachdem im vorangegangen Kapitel Lösungsansätze diskutiert wurden, um die Ziele dieser Arbeit zu erfüllen, behandelt dieses Kapitel die Umsetzung dieser. Im Zuge dessen werden qualitätssichernde Maßnahmen in einer Testumgebung erprobt, welche in Form einer virtuellen Maschine besteht. Dies umfasst die einzelnen Testphasen zu planen und durchzuführen. Darüber hinaus soll ein weitergehendes Konzept erarbeitet werden, was den Schlusspunkt dieses Kapitels bildet.

5.1 Unit-Tests speziell in TYPO3

Um einen Unit Test für eine TYPO3 Extension zu schreiben, ist es nicht ausreichend einen herkömmlichen PHPUnit Testcase zu erstellen. Damit sich die Extension auch innerhalb des Tests wie gewünscht verhält, ist es notwendig, die Basisumgebung von TYPO3 zu initialisieren. Es müssen beispielsweise Variable konfiguriert und bereitgestellt werden, auf die jede Extension unweigerlich zugreift. Um dies zu bewerkstelligen, gibt es eine Extension, welche PHPUnit für TYPO3 bereitstellt. Diese setzt auf der nativen PHPUnit auf und erweitert diese um ein Interface und die benötigten Funktionen, um Unit-Tests innerhalb von TYPO3 ausführen zu können.

Beim Erstellen eines Tests ist darauf zu achten, dass die Elternklasse, von welcher geerbt wird, nicht länger aus dem PHPUnit Framework sondern aus der Extension stammt. Dadurch ist es nicht mehr möglich, den Test auf der Kommandozeile auszuführen, da das PHPUnit Framework den Testfall nicht mehr als solchen erkennt. Das würde bedingen, dass die Tests nicht automatisiert durch einen Continous Integration Server ausgeführt werden können. TYPO3 bietet für diese Problematik einen CLI Dispatcher. Dieser initialisiert die TYPO3 Umgebung und ermöglicht es, dass Extensions auf der Kommandozeile angesprochen werden können. Infolgedessen lassen sich Tests, welche mit der PHPUnit Extension erstellt wurden, ebenfalls auf der Kommandozeile ausführen. Das Verhalten gleicht dabei einer herkömmlichen Testausführung, womit diese Tests in einen Continous Integration Zyklus implementiert werden können.

Nicht zu unterschätzen ist der Nutzen der grafischen Oberfläche. (vgl. Abbildung 5–1) Das oft zitierte Rot-Grün-Gefühl von JUnit ist in der PHPUnit Umgebung unbekannt, wenn es in einem Windows System betrieben wird. Da Tests nur auf der Kommandozeile ausgeführt werden, werden die Testergebnisse lediglich in Textform publiziert. Eclipse und ähnliche

Programmierumgebungen bieten ebenfalls nur eine limitierte grafische Unterstützung. Somit ist das Interface eine zweckdienliche Erweiterung, um das Schreiben von Tests zu befördern. Denn nicht zu vergessen ist der psychologische Effekt, der von Signalfarben wie Rot und Grün ausgeht. Eine Testsuite, welche im „grünen Bereich“ ist, schafft Vertrauen in das Erreichte und gibt sofortige Sicherheit, dass keine Fehler im Quellcode auftreten. Dies wiederum bestärkt Entwickler Tests zu schreiben, um diese Sicherheit zu gewährleisten.

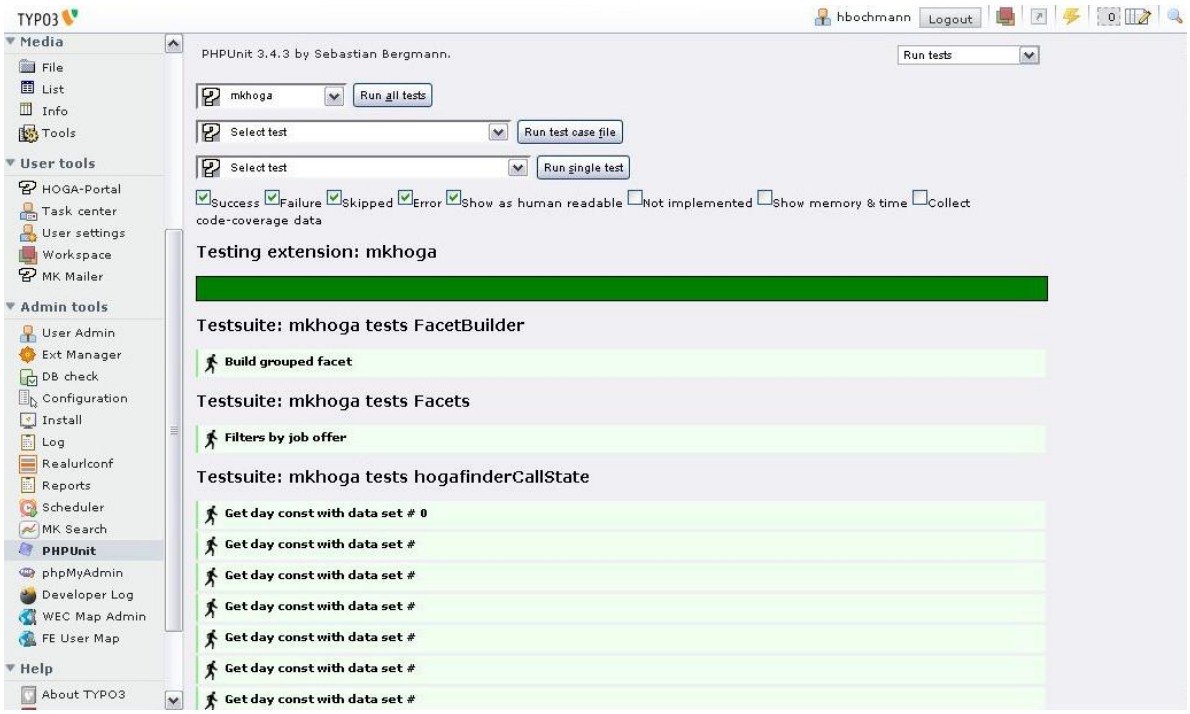


Abbildung 5–1: TYPO3 Extension PHPUnit

5.2 Die Planungsphase

Die Testarbeiten innerhalb des Hoga Jobportals werden nach dem allgemeinen V-Modell mit Aspekten testgetriebener Entwicklung integriert. (vgl. Kapitel 4) Dementsprechend müssen 3 Testphasen geplant werden. Die Planung soll im Folgenden analysiert werden.

5.2.1 Komponententest

Die Analyse in Kapitel 4 hat ergeben, dass im Projekt des Hoga Jobportals nach dem Modell der testgetriebenen Entwicklung vorgegangen werden soll. Dabei bietet ein Continous Integration Server wertvolle Unterstützung für die Testautomatisierung. Dementsprechend wird ein CruiseControl Server in Verbindung mit phpUnderControl betrieben. Dieses Zusammenspiel bietet große Vorteile, da eine fortlaufende Integration und automatisierte Testausführung gewährleistet wird.

Bisher wurde das Deployment mit Hilfe eines Ant Skriptes manuell in einer IDE durchgeführt. Dieses Skript wird nun als Build Skript für den CruiseControl Server verwendet, um den Deploymentprozess weiter zu automatisieren. Dabei muss das Skript um die Ausführung diverser Werkzeuge erweitert werden. Dazu zählt zum einen PHPUnit, welches die Tests ausführt, ein Log generiert und die Testüberdeckung analysiert (vgl. Kapitel 4.4). Zum anderen PHPDoc, welches eine Dokumentation erstellt, und PHPMD, welches die Codequalität überprüft. Die Codequalität wird u.a. durch das Auftreten von duplizierten Code oder Initialisierung von nicht verwendeten Variablen definiert. Zu beachten ist hierbei, dass diese Werkzeuge sowohl auf der Umgebung, in welcher das Build ausgeführt wird, als auch remote funktionieren müssen.

Bei den Unit-Tests handelt es sich um funktionale und änderungsbasierte Tests, da bei jeder Änderung im SVN Repositorium und somit bei jeder Änderung am Code, ein Build ausgeführt wird. Die automatisch generierten Testlogs entsprechen dabei der Testdokumentation nach IEEE 829 (vgl. Abschnitt 5.4.2). Die Tatsache, dass diese über den gesamten Projektverlauf hinweg archiviert werden, macht eine Wiederholungstestmatrix überflüssig (vgl. Kapitel 2.2.3.3)

Der CruiseControl Server ist konfiguriert, im Fall eines fehlgeschlagenen Builds eine Email zu verschicken. Diese wird hierbei an den Server Administrator und an den Entwickler, welcher zuletzt Code in das SVN Repositorium übertragen hat, versendet. Die Oberfläche des Servers bietet nicht nur Informationen hinsichtlich der bestandenen Tests oder der Ergebnisse von PHPMD, welche für Entwickler wertvoll sind. Ergebnisse zur Anzahl von fehlgeschlagenen Builds, der Fehlerrate oder der erreichten Testabdeckung sind ebenfalls für Projektmanager oder Kunden interessant, um einen präzisen Projektüberblick zu erhalten.

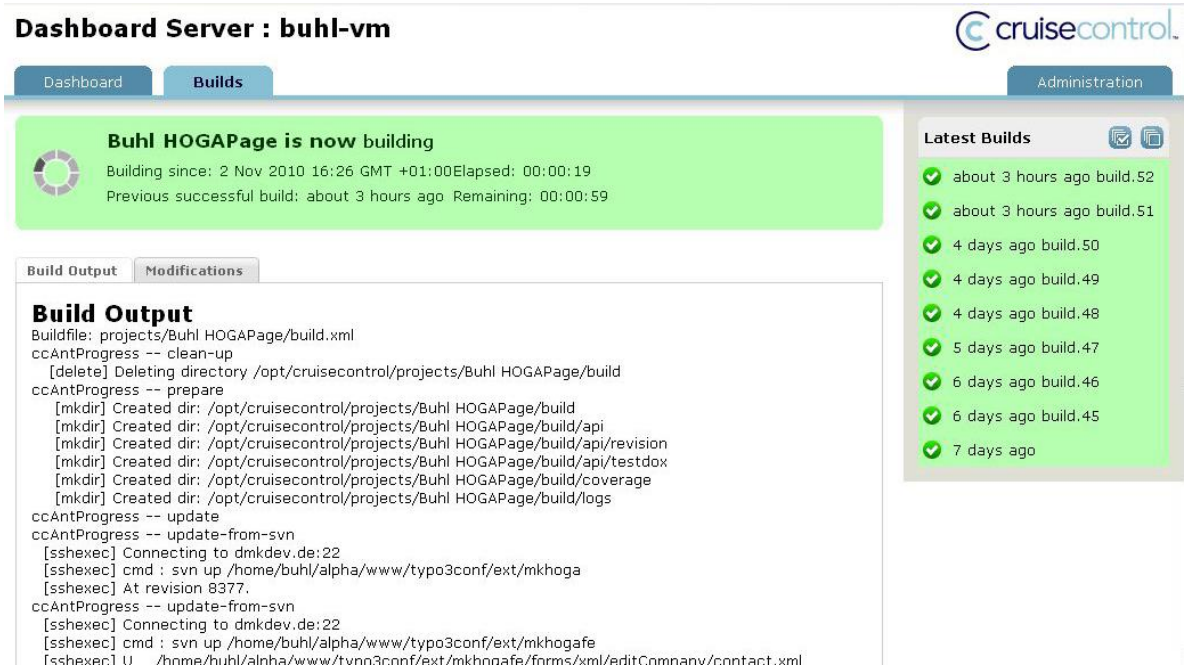


Abbildung 5–2: Build des CruiseControl Servers

Mit Hilfe des Continous Integration Servers und der damit verbundenen Maßnahmen werden die Software Qualitätsmerkmale Änderbarkeit, Zuverlässigkeit, Funktionalität und Effizienz geprüft und bewertet. Änderbarkeit und Zuverlässigkeit werden daran gemessen, wie viele Tests und damit auch Builds in der Projekthistorie fehlgeschlagen sind. Dies gilt insbesondere im Zuge von größeren Anpassungsmaßnahmen. Die Funktionalität und Effizienz wird anhand der Testberichte und Ergebnisse von PHPMD beurteilt.

Die Konfigurations- und Builddateien sind im Anhang zu finden.

5.2.2 Integrations-/Oberflächentest

In Kapitel 4.6.3 wurde analysiert, dass Capture and Replay Werkzeuge notwendig sind, um Cross Browser Tests in der Integrations- und Oberflächenteststufe effektiv zu erstellen. Demzufolge werden diese Tests mit Hilfe der Selenium Softwaresuite generiert. Das Konzept sieht dabei vor, Tests, wie das Erstellen einer Anzeige, über die Selenium IDE aufzuzeichnen und als PHPUnit Testcase zu exportieren, womit sie durch den CruiseControl Server automatisiert ausgeführt werden können. Damit die exportierten Tests ausführbar sind, muss eine Erweiterung für PHPUnit installiert werden. Nach dem Export werden die Tests mit Prüfungen der Geschäftslogik angereichert. Das umfasst u.a. korrekte Datenbankeinträge oder die richtige Belegung von TYPO3 Umgebungsvariablen.

Damit die Tests ausgeführt werden können, werden weiterhin Selenium Remote Controls innerhalb eines Grids und ein Hub benötigt. Sowohl das Selenium Grid als auch die Remote Controls werden auf einer dedizierten virtuellen Maschine auf Basis von Windows XP betrieben. Die bereitgestellten Browser umfassen dabei die gängigen, wie Firefox, Internet Explorer oder Safari. Der Hub läuft auf der gleichen Maschine wie der CruiseControl Server. Die gesamte Testsuite wird in Form eines eigenen CruiseControl Builds jede Nacht ausgeführt und am nächsten Tag ausgewertet.

Die Tests sind dabei sowohl funktionale Tests, da Geschäftslogik geprüft wird, als auch nicht-funktionale, da die Korrektheit der Oberfläche geprüft wird (vgl. Kapitel 2.2.3). Somit bewerten diese Tests die Qualitätsmerkmale Funktionalität und Benutzbarkeit.

5.2.3 Last-/Performanztest

Die Last- und Performanzteststufe stellt, auf Grund des Zeitdrucks im Projekt, nur eine sehr kurze Phase dar. Sie dient dazu, gravierende Mängel, auf Grund von Fehlern in der Serverkonfiguration oder insuffizientem Programmcode, zu finden. Daher wird lediglich JMeter als Testwerkzeug eingesetzt und Testfälle müssen risikoorientiert (vgl. Kapitel 5.4.7) entworfen werden. Die 3 wichtigsten Szenarien, welche aufgezeichnet und als Basis für die Leistungsermittlung herangezogen werden, sind jeweils ein Test mit den Benutzergruppen Arbeitnehmer, Arbeitgeber und anonym. Im Fall der Arbeitnehmer und Arbeitgeber werden Nachrichten im Postfach gelesen und die Stammdaten bearbeitet. Das Szenario der anonymen Nutzergruppe umfasst das Aufrufen allgemeiner Informationsseiten und das mehrfache Abschicken von Suchanfragen. Zusätzlich ist es notwendig, einen Test mit dem Aufruf einer statischen Html Datei durchzuführen, um einen Referenzwert für die maximale Leistung des Systems zu ermitteln.

Da keine längere Laufzeit der Tests geplant ist, d.h. nicht über mehrere Wochen, werden diese manuell angestoßen und nicht in den Continous Integration Zyklus integriert. Dabei werden die Qualitätsmerkmale Zuverlässigkeit und Effizienz geprüft und bewertet.

5.3 Die Realisierungsphase

Entsprechend der Planungen im vorherigen Abschnitt wird der Testprozess im Projekt des Hoga Jobportals gestaltet. Dieser Abschnitt erläutert anhand von gekürzten Beispielen wie die Planung umgesetzt wurde.

5.3.1 Komponententest

In Kapitel 4 wurden die Grundlagen vermittelt, welche nötig sind, um testgetriebene Entwicklung zu verstehen und mit welchem Werkzeug dieser Prozess unterstützt werden kann. Wie testgetriebene Entwicklung im Detail funktioniert wird im Folgenden anhand einer gekürzten und vereinfachten Programmierepisode erläutert. Dabei wird die TYPO3 Extension phpunit als Testframework verwendet. Wie bereits erwähnt sind die Tests funktional, anforderungsbasiert und den Greybox-Verfahren zuzuordnen.

Die zu erfüllende Anforderung ist nun, einen Validator bereitzustellen, welcher prüft, ob eine Anzeige erst in der Zukunft geschalten wird. Somit ist zuerst ein Test zu schreiben, der diese Anforderung prüft. Allerdings wird an dieser Stelle auf eine Grenzwertanalyse (vgl. Kapitel 2.3.1.2) bzgl. der Testeingabedaten verzichtet. Eine Äquivalenzklassenbildung ist ausreichend.

Zu Beginn muss die Testklasse aus dem PHPUnit-Framework abgeleitet werden. Um die Eingabedaten für den Test bereitzustellen, bietet es sich an, einen Data Provider zu schreiben. Dieser beinhaltet die Repräsentanten der Äquivalenzklassen und bildet den Startpunkt.

```
1  class models_JobAdTest extends tx_phpunit_testcase {
2        public function dataProvider(){
3              return array(
4                           array('1 days',
5                                 'JobAd sollte in Zukunft sein'),
6  /*                       array('0 days',
7                                 'JobAd sollte nicht in Zukunft sein'),
8                           array('-1 days',
9                                 'JobAd sollte nicht in Zukunft sein'),
10 */
11                          );
12 }
13 }
```

Es wurden bereits alle Repräsentanten implementiert, aber es werden noch nicht alle verwendet, um die Entwicklung inkrementell gestalten zu können. Bei den nicht verwendeten handelt es sich um die ungültigen Äquivalenzklassen. Bei Implementierung aller Repräsentanten zu Beginn würde es u.U. lange dauern bis der Test bestanden wird, da zu viele verschiedene Anforderungen erfüllt werden müssen. Der Gedanke der testgetriebenen Entwicklung ist, jede Anforderungen in kleinere Teilanforderungen zu unterteilen, welche schnell erreichbar sind. Anschließend wird für jede Teilanforderungen

ein Test geschrieben. Jeder bestandene Test bildet einen Punkt, an dem die getestete Funktionalität als erreicht und gesichert gilt.

Die erste Teilanforderung umfasst, dass die zu testende Funktion den boolean Wert „wahr“ zurück gibt, wenn eine Anzeige in der Zukunft beginnt und endet.

Zunächst muss die Anzeigen-Klasse mit den zu testenden Eingabedaten im Testfall initialisiert werden. Der eigentliche Test geschieht durch den Aufruf der Methode assertTrue, in welcher der Rückgabewert der isFuture Methode überprüft wird.

```
1  /**
2  * @dataProvider dataProvider
3  */
4  public function test_isFuture($a,$b) {
5         $jobAd = new models_JobAd($a);
6         $this->assertTrue($jobAd->isFuture(),$b);
7  }..
```

Nun ist der erste Testfall geschrieben und kann ausgeführt werden.

```
hbochmann@ubuntu ~ % phpunit -verbose models_JobAdTest
PHPUnit 3.4.11 by Sebastian Bergmann
models_JobAdTest
E
Fatal Error: Call to undefined method models_JobAd::isFuture() in…
```

Wie zu erwarten schlägt der Test fehl, da die Funktion noch nicht existiert. Also sollte diese zunächst geschrieben werden und zwar so einfach wie möglich. Die Lösung im nächsten Schritt scheint nicht sinnvoll, ist aber die einfachste, die möglich ist, damit der Test bestanden wird. Und das sollte das Hauptaugenmerk sein, um schnell zum Teil des Refactorings vordringen zu können, was die eigentliche Programmierarbeit darstellt. Besonders in der Einarbeitungsphase mit testgetriebener Entwicklung bietet es sich an, in solch kleinen Schritten voran zu gehen, um den Nutzen zu verstehen.

```
1  class models_JobAd{
2         public function isFuture() {
3                return true;
4         }
5  }
```

Wird der Testfall nun ausgeführt, wird er bestanden. Aber es wird bis jetzt nur eine Möglichkeit von Eingabedaten getestet. Also muss der Testfall um eine sinnvolle Auswahl erweitert werden, bei der die Anzeige am aktuellen Tag beginnt. Dementsprechend wird der Data Provider angepasst und das zweite Eingabedaten-Array ebenfalls zurückgegeben. Der

Testfall wird dabei so lang ausgeführt wie der Data Provider Daten zurückgibt, in diesem Fall zweimal.

Beim erneuten Ausführen schlägt der Test wieder fehl. Die ausgegebenen Fehlernachrichten können dabei hilfreich sein, zu identifizieren, was fehlgeschlagen ist. In diesem Fall sollten die neu hinzugefügten Testdaten bewirken, dass die Funktion „falsch“ zurück gibt, da die Anzeige nicht in der Zukunft beginnt. Bis jetzt sah der Weg des Tests wie folgt aus: Fehlgeschlagen → Bestanden → Fehlgeschlagen. Nun muss die zu testende Funktion angepasst werden, damit der Test wieder bestanden wird.

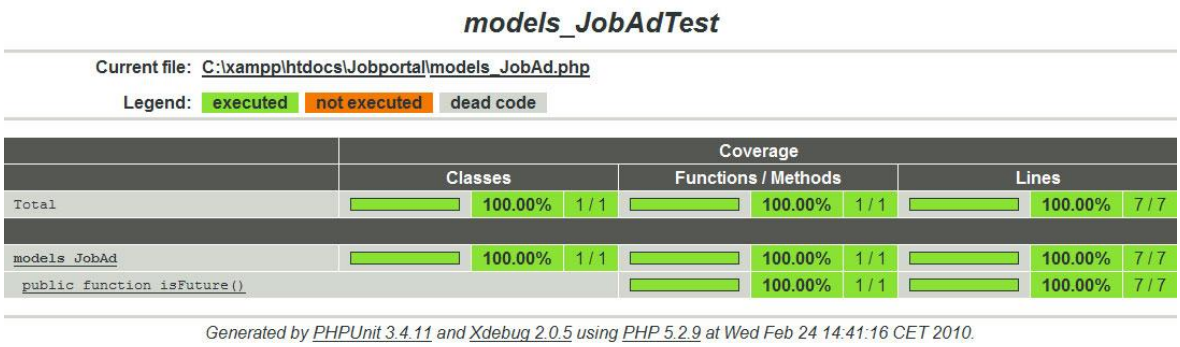
```
1  public function isFuture() {
2          if(substr($this->from,0,1) == 1)
3                  return true;
4          else
5                  return false;
6  }
```

Der Test wird nun wieder bestanden. Da aber die Auswahl an Testdaten noch nicht ausreichend und die Funktion sehr statisch ist, ist ein Refactoring von Nöten, um die Funktion dynamisch auf Eingabedaten reagieren zu lassen. Denn wird als Offset „10 Days“ oder “+1 Days“ angegeben, funktioniert die Funktion nicht mehr wie gewünscht. Aber bevor die Testeingabedaten erweitert werden findet das Refactoring statt. Dies sollte immer geschehen wenn der Test bestanden ist, um keinen Fortschritt zu verlieren. Bis jetzt funktioniert alles mit den Eingabedaten wie erwartet und das sollte auch nach dem Refactoring der Fall sein. Die Funktion könnte wie folgt angepasst werden.

```
1  public function isFuture() {
2          $today = date('Y-m-d');
3          $given = new DateTime();
4          $given->modify($this->start);
5          return ($given->format('Y-m-d') > $today)
6  }
```

Wird der Testfall nun ausgeführt, wird er noch immer bestanden. Nun sollten die Testeingabedaten komplettiert und das gesamte Array des Data Providers zurück gegeben werden.

Zusätzlich kann ein Code Coverage Report erstellt werden, um nachzuweisen, dass die Funktion vollständig durchlaufen wird. (vgl. Abbildung 5–3)



models_JobAdTest

Current file: C:\xampp\htdocs\Jobportal\models_JobAd.php

Legend: executed | not executed | dead code

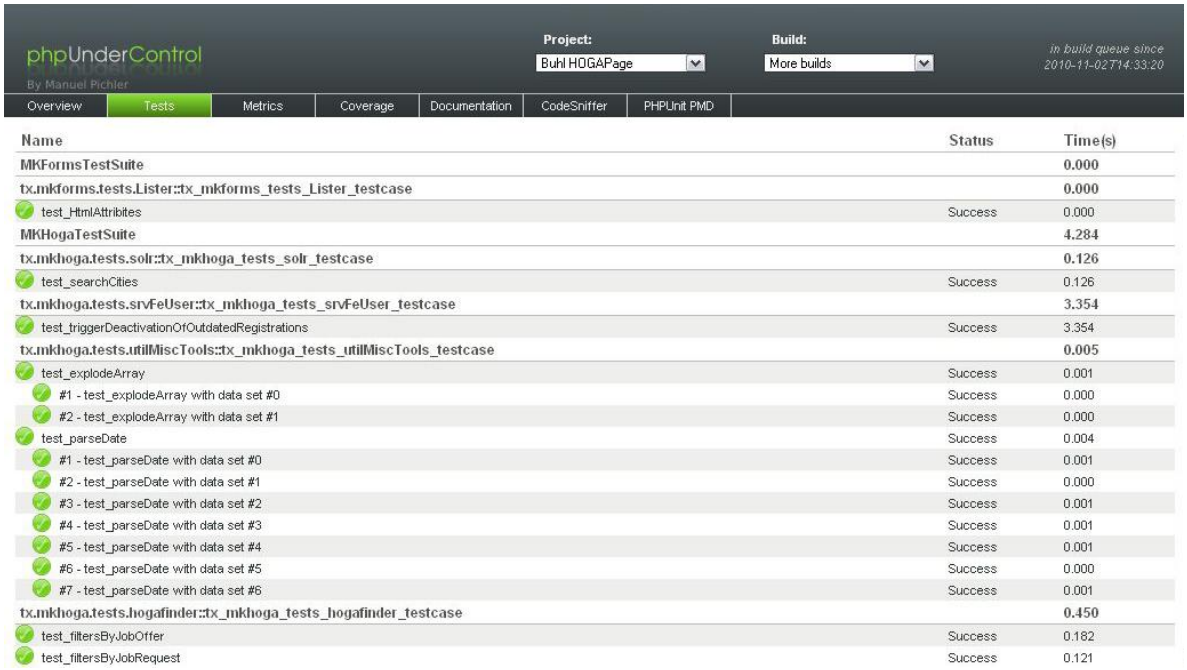
	Coverage								
	Classes			Functions / Methods			Lines		
Total		100.00%	1 / 1		100.00%	1 / 1		100.00%	7 / 7
models_JobAd		100.00%	1 / 1		100.00%	1 / 1		100.00%	7 / 7
public function isFuture()					100.00%	1 / 1		100.00%	7 / 7

Generated by PHPUnit 3.4.11 and Xdebug 2.0.5 using PHP 5.2.9 at Wed Feb 24 14:41:16 CET 2010.

Abbildung 5–3: Code Coverage Report

Nach dieser Episode ist die Funktion komplett durch Tests abgedeckt, welche die Richtigkeit der Funktion nachweisen. Eventuell sind die Testeingabedaten nicht vollständig aber dies ist an dieser Stelle zu vernachlässigen. Zusätzlich sind zukünftige Refactorings einfach, da die erreichte Funktionalität durch die Tests abgesichert ist.

Natürlich ist es z.B. unvorteilhaft, das Startdatum direkt in der Funktion zu berechnen und den Offset-Wert in einer Klassenvariablen zu speichern. Ein weiterer Refactoringschritt wäre, das Datum im Konstruktor zu berechnen oder beispielweise formatiert aus einer Datenbank zu übergeben. Dies soll aber nicht näher betrachtet werden, da dieses Beispiel lediglich die Grundzüge testgetriebener Entwicklung veranschaulichen soll. In der Praxis wurden viele weitere Testfälle erstellt. (vgl. Abbildung 5–4)



phpUnderControl — By Manuel Pichler

Project: Buhl HOGAPage | Build: More builds | in build queue since 2010-11-02T14:33:20

Overview | Tests | Metrics | Coverage | Documentation | CodeSniffer | PHPUnit PMD

Name	Status	Time(s)
MKFormsTestSuite		0.000
tx.mkforms.tests.Lister::tx_mkforms_tests_Lister_testcase		0.000
test_HtmlAttribites	Success	0.000
MKHogaTestSuite		4.284
tx.mkhoga.tests.solr::tx_mkhoga_tests_solr_testcase		0.126
test_searchCities	Success	0.126
tx.mkhoga.tests.srvFeUser::tx_mkhoga_tests_srvFeUser_testcase		3.354
test_triggerDeactivationOfOutdatedRegistrations	Success	3.354
tx.mkhoga.tests.utilMiscTools::tx_mkhoga_tests_utilMiscTools_testcase		0.005
test_explodeArray	Success	0.001
#1 - test_explodeArray with data set #0	Success	0.000
#2 - test_explodeArray with data set #1	Success	0.000
test_parseDate	Success	0.004
#1 - test_parseDate with data set #0	Success	0.001
#2 - test_parseDate with data set #1	Success	0.000
#3 - test_parseDate with data set #2	Success	0.001
#4 - test_parseDate with data set #3	Success	0.001
#5 - test_parseDate with data set #4	Success	0.001
#6 - test_parseDate with data set #5	Success	0.000
#7 - test_parseDate with data set #6	Success	0.001
tx.mkhoga.tests.hogafinder::tx_mkhoga_tests_hogafinder_testcase		0.450
test_filtersByJobOffer	Success	0.182
test_filtersByJobRequest	Success	0.121

Abbildung 5–4: Testsuite Ergebnisse

5.3.2 Integrations-/Oberflächentest

Nachdem in Abschnitt 5.2.2 erläutert wurde, wie der Testprozess in der Integrations- und Oberflächenteststufe geplant ist, wird in diesem Abschnitt anhand eines Beispiels erläutert, wie dieser in der Praxis gestaltet wird.

Wie bereits erwähnt, werden die Tests zunächst in der Selenium IDE aufgezeichnet und anschließend als PHPUnit Testcase exportiert und erweitert. Es ist anzumerken, dass es sich bei der Testfallermittlung um Blackbox-Verfahren handelt. Diese sind geschäftsprozessbasiert und richten sich nach dem Risiko für den wirtschaftlichen Erfolg des Hoga Jobportals. Ein Testfall ist z.B. die Erstellung einer Text-Anzeige, da dies ein komplexes Modul und essentiell für das Jobportal ist. Alle Testfälle wurden in einem Testprotokoll aufgelistet, um die Erstellung systematisch zu gestalten und sind im Anhang zu finden.

Die Laufzeit der gesamten Testsuite beträgt im Schnitt vier Stunden, wodurch diese effektiv nur in der Nacht ausgeführt werden können. Aufgetretene Fehler werden am darauf folgenden Tag identifiziert und behoben. Die Tests schlagen sporadisch wegen zu langen Antwortzeiten fehl, was nicht Fehlern in der Programmierung geschuldet ist. Allgemein ist zu konstatieren, dass die Tests sehr fragil sind und seltener auf Grund von tatsächlichen Programmierfehlern fehlschlagen. Ein weiterer Grund dafür ist die Verbindung funktionaler und nicht-funktionaler Tests. Dadurch kann ein Test der Oberfläche beispielsweise wegen Problemen mit der Datenbankverbindung fehlschlagen. Daher müssen die Ergebnisse mehrerer Testläufe ausgewertet werden, um tatsächliche Fehler aufzudecken.

Employer::Employer_ManageJobAdsTest		**840.257**
Employer_ManageJobAdsTest: Safari on Windows		**840.257**
testCorrectErrorMessagesAreDisplayed	Success	68.100
testDraftJobAdCanBeDeleted	Success	75.515
testDraftJobAdCanBeWatched	Error »	
testDraftJobAdCanBeChanged	Error »	
testDraftJobAdCanBePublished	Error »	
testPublishedJobAdCanBeWatched	Error »	
testPublishedJobAdCanBeDeactivatedAndReactivated	Success	83.012
testPublishedJobAdCanBeCopied	Failure »	
testPublishedJobAdCanBeExtended	Error »	
testPublishedJobAdCanBeArchived	Error »	
testArchivedJobAdCanBeWatched	Error »	
testArchivedJobAdCanBeReactivated	Error »	
testArchivedJobAdCanBeDeleted	Success	76.985
Employer::Employer_MasterDataTest		**1833.133**
Employer_MasterDataTest: Safari on Windows		**1833.133**
testAccessDataFormContainsCorrectInitialData	Success	90.959
testChangeAccessDataWithInvalidData1Fails	Success	88.430
testChangeAccessDataWithInvalidData2Fails	Success	88.527
testChangeAccessDataWithExistingEmailFails	Success	89.112
testAccessDataIsChangedCorrect	Success	88.210
testCompanyContactDataFormContainsCorrectInitialData	Error »	
testChangeCompanyContactDataWithInvalidDataFails	Error »	
testCompanyContactDataIsChangedCorrect	Error »	
testContactPersonContactDataFormContainsCorrectInitialData	Success	96.741
testChangeContactPersonContactDataWithInvalidDataFails	Success	104.795
testContactPersonContactDataIsChangedCorrect	Success	106.676
testCorrectErrorMessageIsDisplayedWhenNoUserAccountsBeenCreated	Error »	
testCreatingNewUserAccountFailsWithInvalidData	Error »	

Abbildung 5–5: Selenium Testsuite Ergebnis

5.3.3 Last-/Performanztest

Die Performanz- und Lasttestsufe gestaltet sich als eine sehr kurze Phase. Auf Grund des Zeitdrucks bleiben nur wenige Tage um die in Abschnitt 5.2.3 erwähnten Testszenarien aufzuzeichnen und zu prüfen. Damit die Ergebnisse aussagekräftig sind, muss ein Testdurchlauf mindestens 12 Stunden betragen. Ansonsten sind nicht genügend Daten vorhanden. Speicherlecks u.ä. treten ebenfalls erst nach längerer Zeit auf. Nach abschließender Bewertung der Ergebnisse konnten jedoch keinerlei gravierende Mängel aufgedeckt werden.

5.4 Weitergehende Konzeptionierung

Auf Grund des engen Zeitrahmens der Diplomarbeit war es nicht möglich, einen umfassenden Testprozess und weiterführende Qualitätssicherungsmaßnahmen zu etablieren. Zum Beispiel konnten nicht alle Testtypen erprobt werden. Daher ist es notwendig, ein weitergehendes und lückenloses Konzept zu entwerfen, um ein Modell zur langfristigen Qualitätssicherung entwickeln zu können. Somit werden Qualitätssicherungsmaßnahmen für zukünftige Projekte komplett fundiert, wodurch die Projekte besser gesteuert werden können. Es ist anzumerken, dass einige Aspekte dieses Konzeptes bereits umgesetzt wurden und lediglich zusammenfassend genannt werden.

5.4.1 Qualitätssicherungsplan nach IEEE 730

Laut (vgl. [Spi05], S. 172) sollte Testen *„nicht als einzige Maßnahme zur Qualitätssicherung eingesetzt werden, sondern im Verbund mit anderen Qualitätssicherungsmaßnahmen stehen."* Die Norm [IEEE 730] bietet hierfür eine Gliederung der zu beachtenden Aspekte:

1. Zweck und Anwendungsbereich der Software
2. Referenzierte Dokumente
3. Projektorganisation und Management
4. Dokumentation
5. Standards, Verfahren, Konventionen
6. Software-Reviews
7. Softwaretests
8. Problemmelde- und Korrekturverfahren
9. Werkzeuge, Techniken und Methoden
10. Verwaltung der Medien und Datenträger
11. Lieferantenmanagement
12. Qualitätsaufzeichungen
13. Trainigsmaßnahmen
14. Risikomanagement
15. Glossar
16. Änderungsverfahren und Änderungsverzeichnis

Wie in der Gliederung zu erkennen ist, wird die Rolle des Testens lediglich im Qualitätssicherungsplan eingeordnet und grob spezifiziert. Die präzise Testplanung erfolgt im Testkonzept und wird im nächsten Abschnitt näher betrachtet.

5.4.2 Testkonzept nach IEEE 829

Die in Kapitel 2 behandelten Grundlagen des Testens lassen erkennen welch umfassende Aufgabe dem Testen zukommt. Aus diesem Grund ist ein gut strukturiertes Konzept notwendig, um die Aufgabe erfolgreich umzusetzen. Die Planung und Präparierung sollte möglichst mit Projektstart begonnen werden und umfasst u.a.: (vgl. [Spi05], S. 173)

- Festlegung der Teststrategie mit Auswahl angemessener Testmethoden
- Entscheidung über Art und Umfang der Testumgebung und der Testautomatisierung (vgl. Kapitel 4.5)

- Festlegung des Zusammenspiels der verschiedenen Teststufen und Integration der Testaktivitäten mit anderen Projektaktivitäten
- Entscheidung wie Testergebnisse ausgewertet und evaluiert werden
- Festlegung von Metriken zur Messung und Überwachung des Testverlaufs und der Produktqualität sowie Definition der Kriterien für das Testende (vgl. Kapitel 2.3.2)
- Festlegung des Umfangs der Testdokumente, u.a. durch das Bereitstellen von Templates
- Erstellung des Testplans mit Entscheidung wer, was, wann und in welchem Umfang testet
- Schätzung des Testaufwandes und der Testkosten; Aktualisierung von Schätzungen und Plänen im Testverlauf

Um die gewonnenen Erkenntnisse in ein Konzept zu bringen bietet die Norm [IEEE 829] eine Gliederung:

1. Testkonzeptbezeichnung
2. Einführung
3. Testobjekte
4. Zu testende Leistungsmerkmale
5. Leistungsmerkmale, die nicht getestet werden
6. Teststrategie
7. Abnahme- und Testendekriterien
8. Kriterien für Testabbruch und Testfortsetzung
9. Testdokumentation
10. Testaufgaben
11. Testinfrastruktur
12. Verantwortlichkeiten/Zuständigkeiten
13. Personal, Einarbeitung, Ausbildung
14. Zeitplan/Arbeitsplan
15. Planungsrisiken und Unvorhergesehenes
16. Genehmigung/Freigabe

Dieser Plan, der durchaus in abgewandelter Form vorliegen kann, muss über den gesamten Lebenszyklus des Projektes ständig aktualisiert und überarbeitet werden, da die bestehenden Testergebnisse zu neuen Erkenntnissen führen können oder sich Anforderungen ändern.

Dabei ist desweiteren eine Priorisierung der Tests vorzunehmen, da es auf Grund von Zeit-, Personal- oder Budgetengpässen dazu kommen kann, dass nicht alle Tests aus- bzw. durchgeführt werden können. Es müssen trotz allem möglichst viele kritische Bereiche getestet werden, um das bestmögliche Testergebnis zu erreichen. Diese Priorisierung wird im nächsten Abschnitt näher betrachtet.

5.4.3 Planung der Testfälle

Laut (vgl. [Spi05], S. 174) gibt es über die bisher genannten Gründe hinaus einen weiteren Vorteil einer Priorisierung der Testfälle. Hochpriore Testfälle im Testplan werden in der zeitlichen Abfolge zuerst erstellt und zur Ausführung gebracht, womit gravierende Fehlerwirkungen frühzeitig aufgedeckt werden können.

Ein möglicher Katalog von Kriterien für die Priorisierung von Testfälle könnte wie folgt aussehen: (vgl. [Spi05], S. 175)

- Die Nutzungshäufigkeit einer Funktion bzw. Eintrittswahrscheinlichkeit einer Fehlerwirkung beim Betrieb der Software. Dies gilt insbesondere für Testfälle im Bereich des Integrationstests.
- Die Priorität der Testfälle kann in Abhängigkeit von der Priorität der (Kunden-) Anforderungen ausgewählt werden. Speziell für Testfälle in den Stufen des Integrations- und System-/Abnahmetests kann dieses Kriterium ein klares Indiz dafür geben, welche Testfälle eine höhere Wichtigkeit genießen.
- Die Sicht der Entwicklungsabteilung kann ebenfalls zu einer Priorisierung der Testfälle herangezogen werden, da diese gut einschätzen kann, welche Komponenten beim Versagen besonders gravierende Auswirkungen nach sich ziehen. Dieses Kriterium kann hilfreich bei der Priorisierung von Testfällen in der Stufe des Komponenten- aber auch des Abnahmetests sein.
- Ein weiteres aussagekräftiges Kriterium liefert die Systemarchitektur im Hinblick auf die Komplexität von einzelnen Komponenten oder Systemteilen. Dieses Kriterium ist relativ allgemeingültig und kann für die Priorisierung von Testfällen in allen Teststufen herangezogen werden.

Nachdem die Testfälle priorisiert sind, ist festzulegen, wann der Testprozess, entweder für einzelne Testtypen, -stufen oder das gesamte Projekt, als erfolgreich gilt und beendet werden kann. Nach (vgl. [Spi05], S.177) besteht *„ohne klare Testendekriterien […] die Gefahr,*

dass die Testarbeiten zufällig, meist jedoch aus Zeitdruck oder Ressourcenmangel beendet bzw. abgebrochen werden." Mögliche Kriterien könnten z.B. folgende sein:

- Erreichter Testumfang: Code Coverage Report (vgl. Kapitel 4.4), abgedeckte Anforderungen, gelaufene Tests
- Erreichte Produktqualität: Fehlerdichte, Fehlerschwere, Ausfallraten bzw. Zuverlässigkeit des Testobjekts
- Verbleibendes Risiko: nicht gelaufene Tests, nicht behobene Fehler, unvollständige Anforderungs- oder Codeabdeckung
- Wirtschaftliche Rahmenbedingungen: Kostenrahmen, Projektrisiken, Liefertermine und Marktchancen

Die Bewertung dieser Kriterien führt zu einem messbaren Ergebnis für die Projektleitung und dient als Entscheidungsgrundlage wann das Testen zu beenden ist.

5.4.4 Planung der Testtypen

Da es z.B. bei Webprojekten spezielle Testtypen gibt und auf Grund der Anforderungen nicht immer alle zum Einsatz kommen, ist es hilfreich, zusätzlich die einzelnen Testtypen zu planen und zu priorisieren. Für das Hoga Jobportal Projekt könnten die Testtypen wie folgt priorisiert werden:

Testtyp	Bewertung	Begründung
Tests zur Funktionalität		
Komponententest	Niedrig	Wird vom Entwicklungsteam selbstverantwortlich durchgeführt; kein Codeabdeckungsgrad vereinbart
Integrationstest	Hoch	Integration in das TYPO3 CMS
Funktionaler Systemtest	Hoch	Geschäftsrelevante Funktionen
Link-Test	Mittel	Wenige externe Links vorgesehen
Cookie-Test	Niedrig	Geschäftsrelevante Funktionen setzen Cookies ein (werden durch TYPO3 gesetzt und sind mit Integrationstest größtenteils abgedeckt)
Plugin-Test	Niedrig	Nur zum Lesen von PDF-Dokumenten werden Plugins benötigt
Sicherheitstest	Hoch	Verarbeitung sensibler Daten; Durchführung geschäftsrelevanter Transaktionen
Tests zur Benutzbarkeit		
Content-Test	Hoch	Geschäftsrelevanter Webauftritt
Oberflächentest	Hoch	Unternehmensstandards und Vorgaben aus dem Lastenheft sind einzuhalten
Browser-Test	Hoch	Benutzer könnten jeglichen Browser nutzen; Kompatibilität zu IE ≥ 7, Opera, Safari, Google Chrome, Mozilla Firefox ≥ 3
Usability-Test	Hoch	Kundenzufriedenheit ist ein kritischer Erfolgsfaktor
Zugänglichkeitstest	Nicht relevant	Zugänglichkeit nach BITV nicht explizit gefordert
Auffindbarkeitstest	Niedrig	SEO-Konzept von externen Anbieter
Tests zur Änderbarkeit und Übertragbarkeit		
Code-Analysen	MIttel	Nur stichprobenweise Code-Inspektionen notwendig
Installationstest	Nicht relevant	Anwendung wird installiert übergeben; Keine Client-Installation notwendig
Tests zur Effizienz und Zuverlässigkeit		
Performanz-/Lasttest	Hoch	Verarbeitung von geschäftsrelevanten Transaktionen; auch für Usability ist Performanz ein Erfolgsfaktor
Ausfallsicherheitstest	Niedrig	Verantwortung liegt bei Cluster-Betreiber
Verfügbarkeitstest	Niedrig	24h Verfügbarkeit; wird bereits durch Performanz-/Lasttest geprüft

Tabelle 5-1: Priorisierung von Testtypen (vgl. [Fra07], S. 220f)

In der Planung ist um Zuge dessen ebenfalls zu beachten, welche Testmittel benötigt werden. Diese umfassen sowohl das Testteam, welches das Know-how besitzen bzw. erlangen sollte, um beispielsweise Tests von speziellen Schnittstellen implementieren zu können. Aber auch Checklisten, eventuelle externe Anbieter und die in Kapitel 4.6 betrachteten Testwerkzeuge. (vgl. [Fra07], S. 222ff)

5.4.5 Planung der Teststufen

Das in Kapitel 2.2.1 betrachtete V-Modell zeigt, welche Entwicklungsstufen und somit auch Teststufen in einem Software-Projekt vorhanden sein können. Dabei sollte jede Stufe die Ergebnisse der vorhergehenden evaluieren um ggf. Anpassungen vorzunehmen. Die zeitliche Reihenfolge der Ausführung von Teststufen sieht folgendermaßen aus und kann in einem Testzyklus mehrfach durchlaufen werden:

1. Komponententeststufe
2. Integrationsteststufe
3. Performanz-/Lastteststufe
4. System-/Abnahmeteststufe
5. Betrieb

Laut (vgl. [Fra07], S. 229) handelt es sich bei der Teststufe Betrieb nicht um eine tatsächliche Teststufe. Sie muss aber mit berücksichtigt werden, da Maßnahmen zur Qualitätssicherung mit der Inbetriebnahme nicht abgeschlossen sind.

Jeder dieser Teststufen können verschiedene Testtypen zugeordnet werden, die zum Einsatz kommen. Eine beispielhafte Zuordnung für ein Webprojekt sieht wie folgt aus: (vgl. [Fra07], S. 230ff)

- Komponententeststufe: testgetriebene Entwicklung und damit Abdeckung des Komponententest durch die Entwickler selbst, Code-Analysen, Sicherheitstest, Link-Test, Plugin-Test, Browser-Test, Content-Test, Oberflächentest
- Integrationsteststufe: Plugin-Test, Integrationstest, Usability-Test
- Performanz-/Lastteststufe: Performanz-/Lasttest, Verfügbarkeitstest, Ausfallsicherheitstest

- System-/Abnahmeteststufe: Funktionaler Systemtest, Sicherheitstest, Content-Test, Browser-Test, Usability-Test, Zugänglichkeitstest, Auffindbarkeitstest, Installationstest, Performanz-/Lasttest, Ausfallsicherheitstest, Verfügbarkeitstest
- Betrieb: eine Pilotphase für einen kleinen, ausgewählten Nutzerkreis kann geplant werden. Während des Betriebs muss weiterhin verifiziert werden, dass sich die Performanz nicht verändert hat, externe Links weiterhin valide sind, die vereinbarte Verfügbarkeit erreicht bleibt, die Sicherheit auch bei neu auftretenden Sicherheitslücken und -risiken gewährleistet ist und die Webseite nach wie vor auffindbar ist.

Kommen, wie in der Zuordnung, Testtypen in verschiedenen Teststufen zum Einsatz, werden die gleichen Testfälle zur Ausführung gebracht.

5.4.6 Kosten- und Wirtschaftlichkeitsaspekt

Da der Testprozess einen durchaus hohen, zusätzlichen Zeit- und damit auch Kostenaufwand für ein Softwareprojekt bedeutet, ist es unabdingbar, die entstehenden Testkosten und damit den Testaufwand in der Planungsphase abzuschätzen. Dies gilt umso mehr, da es im Prinzip nie genug Tests gibt. Somit gilt die Abschätzung, ebenfalls dem Testprozess einen Kostenrahmen zu setzen und beantwortet ein Stück weit die Frage, welcher Testaufwand für ein Projekt angebracht und nützlich ist. Nach (vgl. [Spi05], S. 177) muss ebenfalls betrachtet werden, welche Kosten Fehler ohne Tests verursachen. Dabei ist zu beachten, dass die Testkosten keinesfalls die Fehlerkosten übersteigen dürfen damit das Projekt wirtschaftlich bleibt.

Fehlerkosten durch unentdeckte Mängel auf Grund eines eingeschränkten oder gar nicht existenten Testprozesses umfassen direkte Fehlerkosten, indirekte Fehlerkosten und Fehlerkorrekturkosten. Direkte Fehlerkosten sind dabei solche, die dem Kunden auf Grund von Mängeln entstehen, und indirekte solche, die dem Hersteller auf Grund von nicht erfüllten Anforderungen entstehen. Unabhängig davon wie hoch das Risiko eines Fehlers ist, gilt es, diese so früh wie möglich zu identifizieren, da Fehlerkosten über Entwicklungsphasen steigen. Dies liegt zum einen an eventuell verursachten Folgefehlern und zum anderen am Mehraufwand an Korrekturarbeiten bei später Entdeckung. (vgl. [Spi05], S. 178)

Um das Risiko von Fehlern zu reduzieren kann ein Testprozess etabliert werden, dessen Kosten sich ebenfalls durch verschiedene Faktoren ergeben. Nach (vgl. [Spi05], S. 179) zählen dazu der Reifegrad des Entwicklungsprozesses und die Mitarbeiterqualifikation,

Qualitätsziele, die Testinfrastruktur, die Qualität und Testbarkeit der Software sowie die Teststrategie. Dabei lassen sich nur die letzten drei Faktoren kurzfristig beeinflussen wenn sich ein Projekt bereits in der Planungsphase befindet. Die Testinfrastruktur lässt sich nur relativ kurzfristig beeinflussen, vorausgesetzt, es werden Open Source Testwerkzeuge und -umgebungen eingesetzt, wodurch dafür zusätzlich entstehende Kosten wegfallen. Allerdings darf man bei neu eingesetzter Software, ohne Erfahrungen mit dem Umgang, den Einarbeitungsaufwand nicht unterschätzen. Die Qualität und Testbarkeit der Software kann schnell durch die in Kapitel 4.2 betrachtete testgetriebene Entwicklung gesteigert werden und die Teststrategie kann frei gewählt werden. Beispiele für verschiedene Teststrategien werden im nächsten Abschnitt betrachtet.

5.4.7 Teststrategien

Nach (vgl. [Spi05], S. 181f) definiert die Teststrategie, welche Ziele erreicht werden sollen und die zugehörigen Maßnahmen. Das umfasst u.a. Modelle zur Ableitung von Testfällen (vgl. Kapitel 2.2), die Definition von Testendekriterien und Methoden um die Testergebnisse zu evaluieren wie Testabdeckungsverfahren (vgl. Kapitel 2.3.2). Somit muss unter anderem eine Entscheidung bezüglich des im vorherigen Abschnitt betrachteten Kosten- und Wirtschaftlichkeitsaspekt getroffen werden, um den Testaufwand festzulegen.

Dabei ist der Zeitpunkt, an dem dies geschieht, entscheidend und beeinflusst die Teststrategie maßgeblich. Beginnt die Test- bzw. Qualitätssicherungsplanung zeitgleich mit der Projektplanung, ist es zum Beispiel denkbar, die Testarbeiten gemäß des allgemeinen V-Modells (vgl. Kapitel 2.2.1) zu gestalten und dadurch präventiv einzugreifen. Somit ist es möglich, die Entwicklungsarbeiten qualitätssichernd zu begleiten und ein umfangreiches Testsystem zu installieren und zu dokumentieren, welches Fehler frühzeitig entdecken kann. Ist das Softwaresystem hingegen schon teilweise oder komplett fertig, lässt sich die Testplanung anhand einer Dokumentation in Zusammenspiel mit einem Lastenheft oder durch exploratives Testen, d.h. durch Erkunden des Quellcodes, definieren. Dies kann die Testplanung allerdings erheblich erschweren, weshalb der vorbeugende dem reaktiven Ansatz vorzuziehen ist.

Übliche Herangehensweisen und damit Teststrategien sind folgende: (vgl. [Spi05], S. 183)

- Modellbasiertes Testen nutzt abstrakte funktionale Modelle des Testobjektes

- Statisches modellbasiertes Testen verwendet statische Modelle über die Verteilung von Defekten, Ausfallraten der Software oder ähnliches.
- Risikoorientiertes Testen legt das Hauptaugenmerk auf sicherheits-, erfolgskritische oder besonders fehleranfällige Bereiche (vgl. Kapitel 5.2.3)
- Ansätze wie das V-Modell oder IEEE 829 bieten standardkonforme Vorgehensweisen
- Tests und Testumgebungen aus bereits bestehenden Projekten können adaptiert werden
- Experten, z.B. externe Dienstleister, im Bereich Softwaretests können ebenfalls herangezogen werden

Unabhängig von der Wahl der Teststrategie müssen mögliche Projekt- und Produktrisiken in die Entscheidungsfindung mit einbezogen werden. Somit ist in die Testplanung ebenfalls ein Risikomanagement mit einzubeziehen. Ein systematisches Vorgehen dafür bieten z.B. die Normen [IEEE 730] (vgl. Kapitel 5.4.1) und [IEEE 829] (vgl. Kapitel 5.4.2). Dabei müssen Risiken identifiziert, priorisiert und Maßnahmen zur Minderung eingeleitet werden. Eine essentielle Maßnahme ist das Testen, um Risiken und Fehler aufzudecken. Die Behandlung der aufgedeckten Fehler muss indes ebenso geplant werden. Dies wird im folgenden Abschnitt genauer betrachtet.

5.4.8 Fehlermanagement

Nachdem Tests erstellt und ausgeführt wurden, besteht der Bedarf nach einer Möglichkeit, die eventuell gefundenen Fehler in einem aussagekräftigen Testbericht zu protokollieren. Ein ausführlicher Testbericht sollte nach jedem Testzyklus, also z.B. nach einem neuen Build, erstellt werden, um Fehler frühzeitig zurück verfolgen und beheben zu können. Weicht das Ist- vom Soll-Verhalten ab, muss sich der Tester laut (vgl. [Spi05], S. 190) vergewissern, ob es sich um eine tatsächliche Abweichung handelt, der Testfall fehlerhaft entworfen wurde, Fehler in der Testautomatisierung auftraten oder dem Tester Fehler bei der Ausführung unterlaufen sind.

Im Falle der Verwendung eines der in Kapitel 4.4 genannten xUnit-Frameworks, können zusätzliche Fehlermeldung automatisch bei dem Fehlschlagen eines Tests ausgeben werden. Dadurch besteht für ein umfassendes Fehlermanagement weiterhin die Notwendigkeit, dass die Entwickler, welche die Tests erstellen, aussagekräftige Fehlermeldungen integrieren, um die Fehlerursache schneller identifizieren zu können.

Werden Tests automatisiert ausgeführt, muss ein eventuelles Fehlschlagen kommuniziert werden. Ein Beispiel wäre, eine Email an den Testmanager und den Entwickler, der die letzten Änderungen im SVN vorgenommen hat, zu schicken. (vgl. Kapitel 5.2.1)

Nach (vgl. [Spi05], S. 191) wird in einem Projekt üblicherweise eine Fehlerdatenbank angelegt, um aufgedeckte Fehler und Mängel erfassen, kategorisieren und verwalten zu können. Fehler können dabei sowohl von Tests als auch vom Kunden entdeckt werden.

Eine effektive Methode des Fehlermanagements bildet das Zusammenspiel von dem im Kapitel 4.6.1 betrachteten CruiseControl, welches zum Starten eines Testzyklus‘ und dem Generieren bzw. Präsentieren der Testberichte eingesetzt wird, und Active Collab, welches ein Ticket-System bereitstellt, in dem alle vom Kunden gemeldeten Fehler kategorisiert und verwaltet werden.

6 Résumé und Ausblick

„Der schlimmste aller Fehler ist, sich keines solchen bewusst zu sein.“

Thomas Carlyle, Essayist und Historiker(1795 – 1881)

Dieses Zitat illustriert treffend die Signifikanz, Fehler aufzudecken und zu beheben. Im Rahmen der Diplomarbeit wurden verschiedene Untersuchungen durchgeführt, mit

dem Ziel geeignete qualitätssichernde Maßnahmen zu ermitteln, welche die Anforderungen der MedienKombinat GmbH am besten erfüllen. In diesem Kapitel wird eine abschließende Auswertung aller durchgeführten Untersuchungen und Implementierungen vorgenommen und ein Ausblick, hinsichtlich des zukünftigen Nutzens und Einsatzes, gegeben.

6.1 Résumé

Die Thematik dieser Diplomarbeit umfasst primär, geeignete qualitätssichernde Maßnahmen für Softwareprojekte innerhalb der MedienKombinat GmbH zu untersuchen und zu etablieren. Dies umfasst die Konzeption und Erprobung eines Testprozesses.

Welche Grundlagen notwendig sind, um einen effektiven und umfassenden Testprozess zu etablieren, wurde im einleitenden Teil dieser Arbeit betrachtet. Im weiteren Verlauf der Arbeit wurden die Ausgangssituation und der aktuelle Stand des Entwicklungs- und Testprozesses innerhalb der MedienKombinat GmbH analysiert. Weiterhin wurde diskutiert, wie die gewonnenen Erkenntnisse für den Entwicklungs- und Testprozess genutzt werden können. Im Zuge dessen wurden verschiedene Testmodelle und –werkzeuge, hinsichtlich ihrer Eignung für das Erreichen der gestellten Anforderungen, betrachtet. Auf Basis davon fand die Implementierung des Testprozesses statt.

Abschließend wurde eine weitergehende Konzeptionierung erarbeitet, um die erarbeiteten Ergebnisse zukünftig zu vervollständigen.

6.2 Bilanz der Arbeit

Die vorliegende Diplomarbeit bietet viele Möglichkeiten, weitere qualitätssichernde Maßnahmen, für zukünftige Projekte zu etablieren. Somit stellt diese Arbeit ein Fundament für die Etablierung und Weiterentwicklung eines umfassenden Systems zur Software-Qualitätssicherung innerhalb der MedienKombinat GmbH dar.

Durch die Entwicklung des Testprozesses wurden erste qualitätssichernde Maßnahmen eingeführt. Sie unterstützen Entwickler bei der Erstellung, Pflege, Planung und Durchführung der Softwaretests. Somit konnte der zeitliche Aufwand bei Erweiterungen und beim Beseitigen von Fehlerzuständen erheblich reduziert werden.

Mit Hilfe der entstandenen Umgebung kann und konnte bereits die Qualität der Entwicklungsergebnisse evaluiert und damit, aus Sicht des Unternehmens, langfristig verbessert werden.

6.3 Ausblick

Die Anforderung eine weitergehende Konzeptionierung zu erarbeiten, ermöglicht es ein umfassendes Qualitätssicherungssystem innerhalb der MedienKombinat GmbH aufzubauen und zu etablieren. Dies umfasst einen vollständigen Testprozess, der von allen Entwicklern angenommen und umgesetzt wird.

Darüber hinaus wäre es denkbar Projektbesprechungen durch regelmäßige Code-Walkthroughs und Reviews zu erweitern, um den Entwicklungs- und Testprozess transparenter zu gestalten. Desweiteren ist es sinnvoll eine Coding-Richtlinie zu entwickeln. Diese kann anschließend, mit Hilfe von PHP CodeSniffer, innerhalb des Continous Integration Zyklus geprüft werden und gewährt eine höhere Konsistenz des Quellcodes.

A Anhang

Anlagenverzeichnis

A.1 Testklassen und Continous Integration Konfigurationsdateien auf CD-ROM

Die zum Abgabetermin dieser Diplomarbeit verfügbaren Testklassen und Konfigurationsdateien des Cruisecontrol Servers liegen der Diplomarbeit in elektronischer Form auf CD-ROM bei.

Quellenverzeichnis

Literaturverzeichnis

[Boe79] **Boehm, Barry:** Guidelines for Verifying and Validating Software Requirements and Design Specification. North Holland, 1979

[Duv07] **Duvall, Paul:** Continous Integration - Improving Software Quality and Reducing Risks. Boston: Addison-Wesley, 2007

[Fra07] **Franz, Klaus:** Handbuch zum Testen von Web-Applikationen. Berlin Heidelberg: Springer Verlag, 2007

[Het93] **Hetzel, Bill:** The Complete Guide to Software Testing. 2. Auflage - Wiley, 1993

[Mey82] **Meyers, Glenford:** Methodishes Testen von Prorgammen. 7. Auflage 2001, Übersetzung von "The Art of Software Testing", John Wiley 1979 - Oldenbourg, 1982

[Spi05] **Spillner, Andreas; Linz, Tilo:** Basiswissen Softwaretest. 3. überarbeitete und aktualisierte Auflage, korrigierter Nachdruck 2007 - Heidelberg: dpunkt.verlag, 2005

[Tra96] **Trauboth, Heinz:** Software Qualitätssicherung - Konstruktive und analytische Maßnahmen. 2. Auflage - München: Oldenbourg, 1996

[Wal90] **Wallmüller, Ernest:** Software-Qualitätssicherung in der Praxis. Hanser, 1990

[Wei78] **Weizenbaum, Joseph:** Die Macht der Computer und die Ohnmacht der Vernunft. Suhrkamp Verlag, 1978

[Wes06] **Westphal, Frank:** Testgetriebene Entwicklung mit JUnit & Fit. Heidelberg: dpunkt.verlag, 2006

Verzeichnis der Internetquellen

[Glaser] Die 10 dramatischsten Softwarefehler In: http://www.zehn.de. Stand 02.08.2010 URL: http://www.zehn.de/die-10-dramatischsten-softwarefehler-47925-0 (letzter Abruf am 02.08.2010)

[XP] Extreme Programming In: http://www.extremeprogramming.org. Stand 30.08.2010 URL: http://www.extremeprogramming.org/map/project.html (letzter Abruf am 30.08.2010)

[PHPUnit] PHPUnit In: http://www.phpunit.de. Stand 18.10.2010 URL: http://www.phpunit.de/manual/3.5/en/index.html (letzter Abruf am 18.10.2010)

[Metrics] PHPUnit and Software Metrics In: http://sebastian-bergmann.de. Stand 31.10.2010 URL: http://sebastian-bergmann.de/archives/689-PHPUnit-and-Software-Metrics.html (letzter Abruf am 31.10.2010)

[Selenium] Selenium Dokumentation In: http://seleniumhq.org/docs/. Stand 11.11.2010 URL: http://seleniumhq.org/docs (letzter Abruf am 11.11.2010)

[Grid] Selenium Grid In: http://selenium-grid.seleniumhq.org. Stand 11.11.2010 URL: http://selenium-grid.seleniumhq.org/how_it_works.html (letzter Abruf am 11.11.2010)

[Httperf] httperf In: http://www.hpl.hp.com/research/linux/httperf/. Stand 11.11.2010 URL: http://www.hpl.hp.com/research/linux/httperf/ (letzter Abruf am 11.11.2010)

[JMeter] Apache JMeter In: http://jakarta.apache.org/jmeter/. Stand 11.11.2010 URL: http://jakarta.apache.org/jmeter/ (letzter Abruf am 11.11.2010)

Verzeichnis der Normen und Standards

[ISO 9126] ISO/IEC 9126-1:2001, Software Engineering – Product Quality – Part 1: Quality Model

[IEEE 730] IEEE Std 730-2002, IEEE Standard for Software Quality Assurance Plans

[IEEE 829] IEEE Std 829-1998, IEEE Standard for Software Test Documentation

[BITV] Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz *In:* http://www.gesetze-im-internet.de*. Stand 23.08.2010 URL:* http://www.gesetze-im-internet.de/bitv/index.html *(letzter Abruf am 23.08.2010)*

[BGG] Behindertengleichstellungsgesetz In: http://www.wob11.de. Stand 23.08.2010 URL: *http://www.wob11.de/bgg.html* (letzter Abruf am 23.08.2010)

Ehrenwörtliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Studienarbeit selbstständig angefertigt, keine anderen als die angegebenen Quellen benutzt, die wörtlich oder dem Inhalt nach aus fremden Arbeiten entnommenen Stellen, bildlichen Darstellungen und dergleichen als solche genau kenntlich gemacht und keine unerlaubte fremde Hilfe in Anspruch genommen habe.

Chemnitz, 21. Februar 2011